UNIVERSIDAD DE MÁLAGA | uma.es

ROBOTIC CONTROL ARCHITECTURES

José Raúl Ruiz Sarmiento
Departamento de Ingeniería de Sistemas y Automática
Based on Material by Javier González Monroy and Cipriano Galindo Andrades

LECTURE CONTENTS

1. THE CONCEPT
   Definition, in context, desirable features.

2. CLASSIFICATION
   Deliberative, Reactive and Hybrid paradigms.

3. ROS
   Evolution, Relevance, Dissemination, Installation and Help, ROS vs ROS 2.

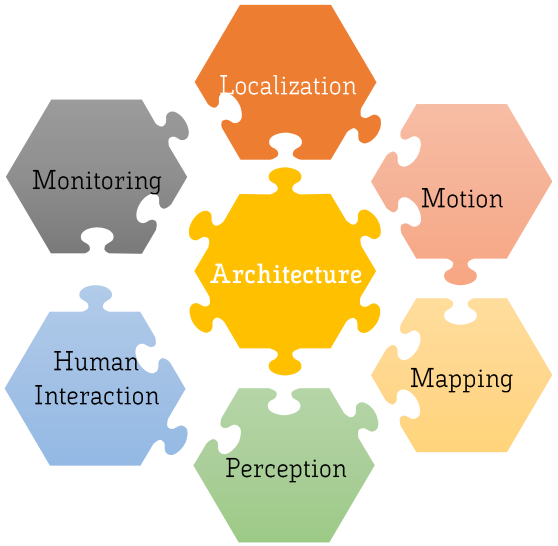
4. ROS 2 – COMPONENTS AND COMMANDS
   Packages, Nodes, Topics, Services, Parameters, Actions.

5. ROS 2 – DEVELOPING SOFTWARE
   Colcon, Workspaces, Creating packages and nodes (CMake and python).

1. THE CONCEPT

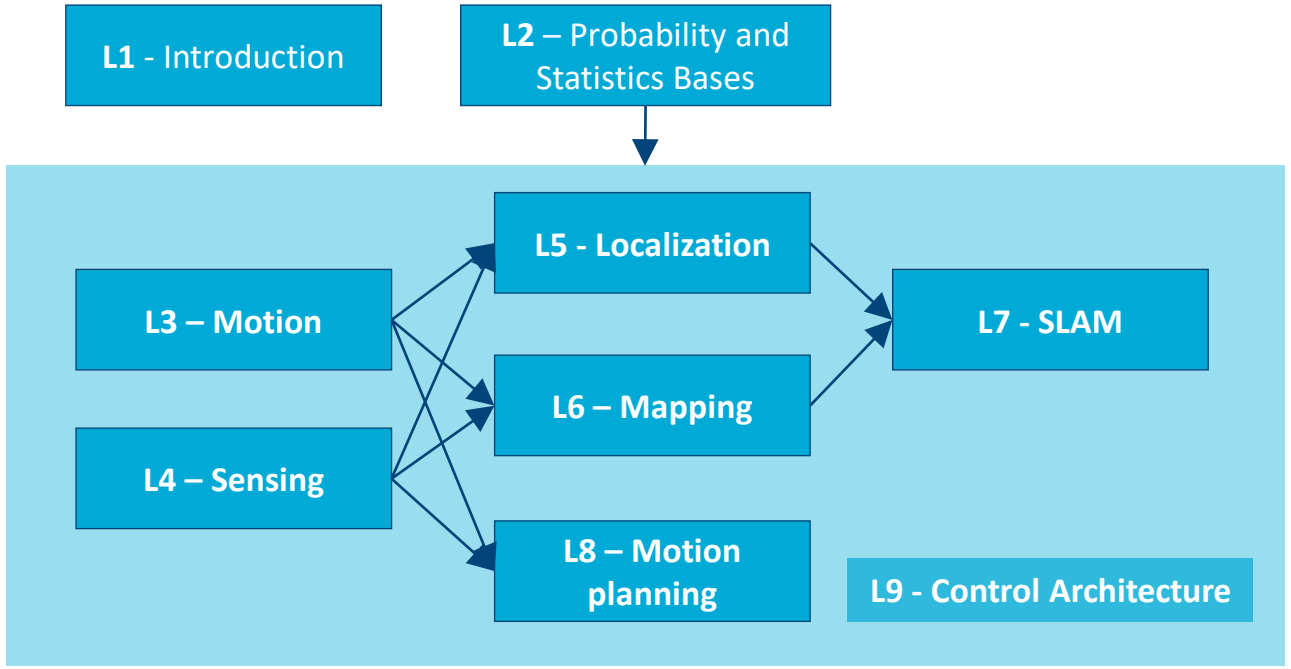
1. THE CONCEPT
DEFINITION

A **Robot Control Architecture** is a software framework that allows the design and implementation of robotic programs/applications from a collection of common building blocks..



1. THE CONCEPT
IN CONTEXT

In the context of our subject:



1. THE CONCEPT
DESIRABLE FEATURES

- **Parallelism**: the ability to execute parallel processes/behaviors at the same time.
- **Run-time flexibility**: does the architecture allow run-time adjustment and reconfiguration? It is important for adaptation/learning.
- **Modularity**: how does the architecture address encapsulation of control, how does it treat abstraction? Does it allow re-use of software?
- **Robustness**: how well does the architecture perform if individual components fail? How well does it enable and facilitate writing controllers capable of fault tolerance?
- **Ease of use**: how easy to use and accessible is the architecture? Are there appropriate programming tools?
- **Performance**: how well does the robot perform using the architecture? Does it act in real-time? Does it get the job done? Is it failure-prone?

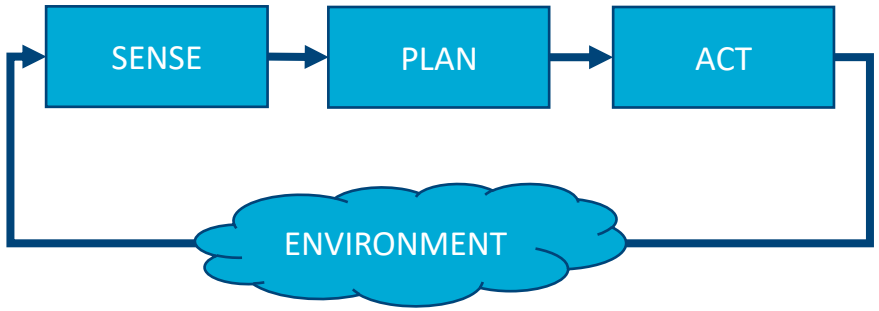
1. THE CONCEPT
DESIRABLE FEATURES

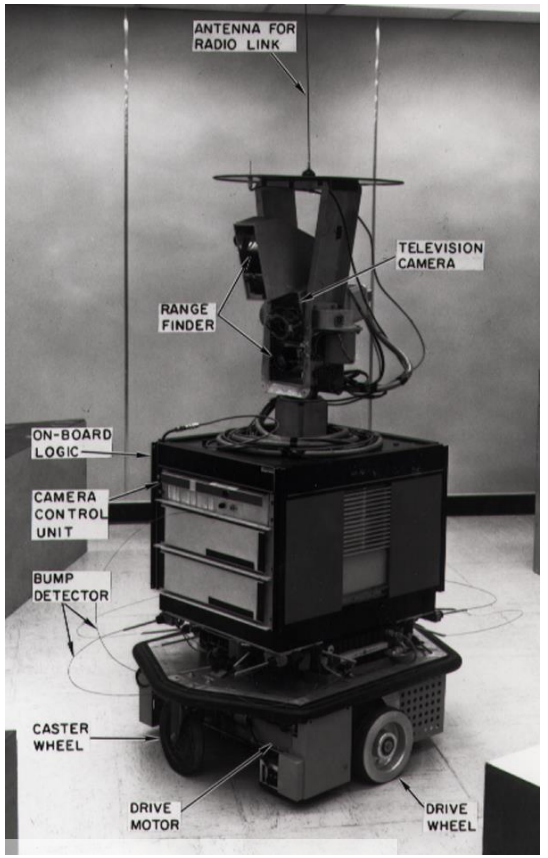
- What should it provide?
  - Communication between modules.
  - Execution monitoring.
  - Sequencing and priorities handling.
  - Sensor reading and processing.
  - Actuation on physical devices.
  - Interfacing with human, internet, other robots, …
  - Scalability.
  - Etc. (depending on the final application)

2. CLASSIFICATION

2. CLASSIFICATION
DELIBERATIVE PARADIGM

- Paradigm of the very first "intelligent" robot Shakey (1967).
- Assumptions:
  - Static Environment.
  - Task execution is perfectly reliable.
  - No reaction against external disturbances.

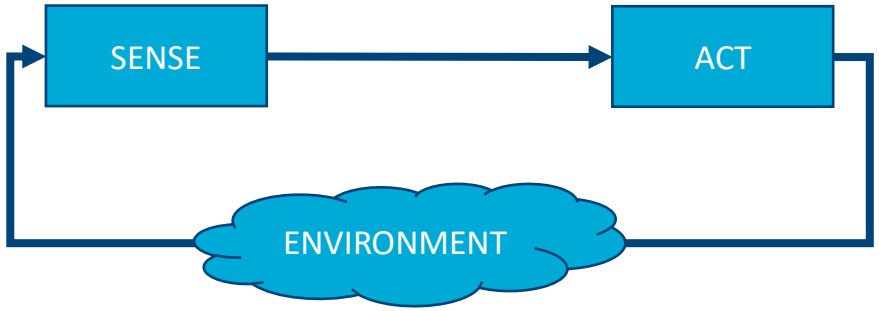




More noticeable results: [video]
- A* search algorithm
- Hough transform
- Visibility graph

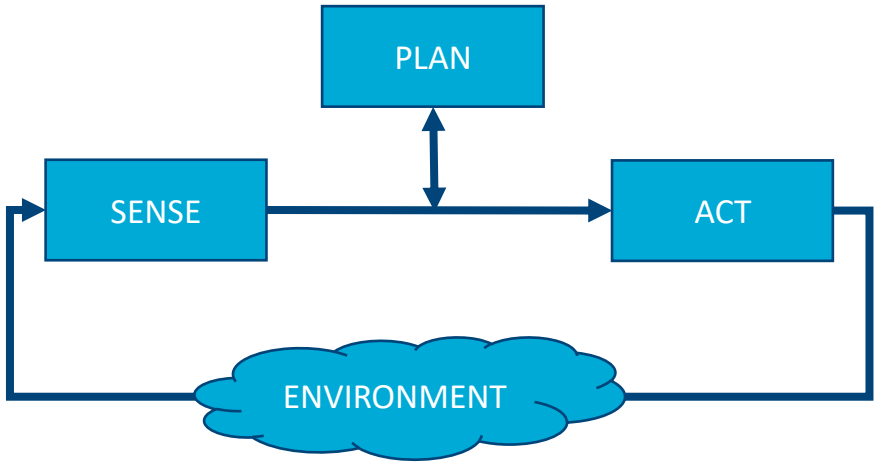
2. CLASSIFICATION
REACTIVE PARADIGM

- Proposed by Michael Arbib in 1980.
- Assumptions:
  - No task planning.
  - Environment highly dynamic and not trusty.
  - Robot operates following reactions.



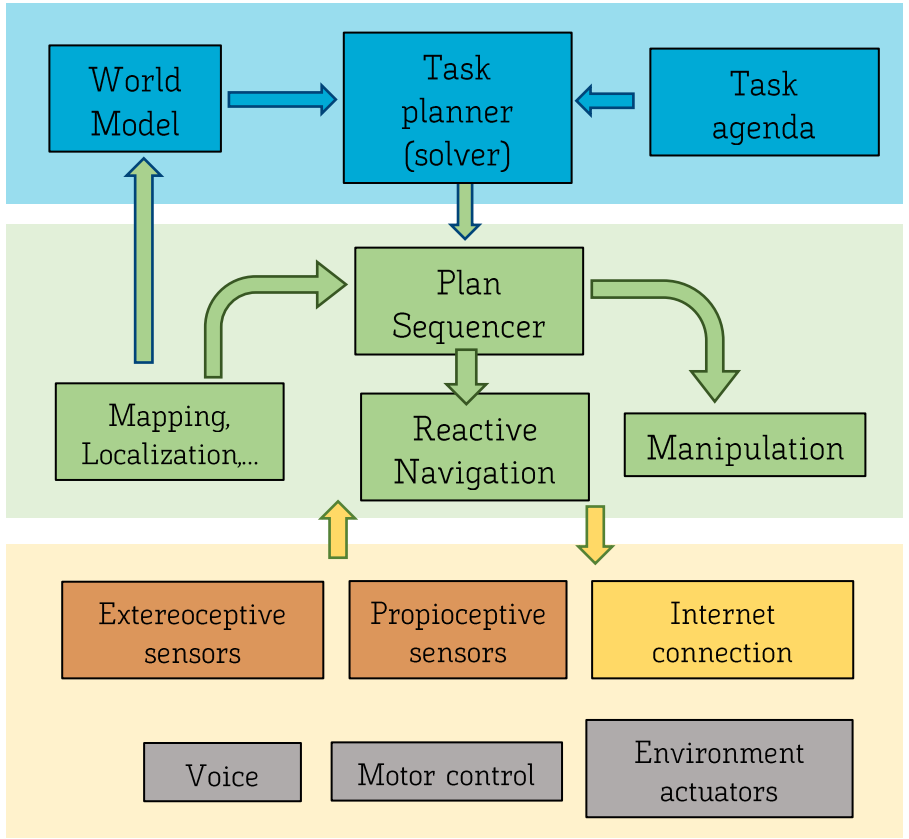
2. CLASSIFICATION
HYBRID PARADIGM

- Combination of previous paradigms (beginning of 1990's)
- Assumptions:
  - Task planning to achieve global objectives.
  - Plan execution monitoring. Reacts against external stimuli.
  - The most commonly adopted nowadays.



2. CLASSIFICATION
HYBRID PARADIGM

- Example:



3. ROS

3. ROS
ROBOT OPERATING SYSTEM

ROS

[video] *Introduction*
[video] *10 years of ROS*

- ROS is an open-source, meta-operating system for your robot.
- It provides: hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, package management, etc.
- Empowers collaborative robotics software development.

3. ROS
ROBOT OPERATING SYSTEM

Collaborative

ROS.org

About | Support | Discussion Forum | Service Status | Q&A answers.ros.org

Search: Submit

Documentation | Browse Software | News | Download

rf2o_laser_odometry

Only released in EOL distros: indigo

Documentation Status

Package Summary

Continuous Integration | Documented

Estimation of 2D odometry based on planar laser scans. Useful for mobile robots with innacurate base odometry. For full description of the algorithm, please refer to: Planar Odometry from a Radial Laser Scanner. A Range Flow-based Approach. ICRA 2016
Available at: http://mapir.isa.uma.es/mapirwebsite/index.php/mapir-downloads/papers/217

- Maintainer status: maintained
- Maintainer: Javier G. Monroy <jgmonroy AT uma DOT es>
- Author: Mariano Jaimez <marianojt AT uma DOT es>, Javier G. Monroy <jgmonroy AT uma DOT es>
- License: GPL v3
- Source: git https://github.com/MAPIRlab/mapir-ros-pkgs.git (branch: master)

Package Links
Code API
FAQ
Change List
Reviews
Dependencies (7)

Tabla de Contenidos
1. Algorithm
2. Demo Video
3. Nodes
    1. rf2o_laser_odometry
        1. Subscribed Topics
        2. Published Topics
        3. Parameters

Algorithm

RF2O is a fast and precise method to estimate the planar motion of a lidar from consecutive range scans. For every scanned point we formulate the range flow constraint equation in terms of the sensor velocity, and minimize a robust function of the resulting geometric constraints to obtain the motion estimate. Conversely to traditional approaches, this method does not search for correspondences but performs dense scan alignment based on the scan gradients, in the fashion of dense 3D visual odometry. The minimization problem is solved in a coarse-to-fine scheme to cope with large displacements, and a smooth filter based on the covariance of the estimate is employed to handle uncertainty in unconstraint scenarios (e.g. corridors).

The user is advised to check the related papers (see here) for a more detailed description of the method.

ROS 2 Documentation

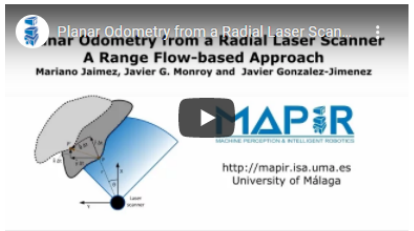
The ROS Wiki is for ROS 1. Are you using ROS 2 (Dashing/Foxy/Rolling)? Check out the ROS 2 Documentation

Wiki
Distributions
ROS/Installation
ROS/Tutorials
RecentChanges
rf2o_laser_odometry

Página
Página inmutable
Información
Adjuntos
Más Acciones
Texto sin formato
Hacer

Usuario
Ingresar

Demo Video



Nodes

0.1 rf2o_laser_odometry

The `rf2o_laser_odometry` node publishes planar odometry estimations for a mobile robot from scan lasers of an onboard 2D lidar. It initially estimates the odometry of the lidar device, and then calculates the robot base odometry by using `tf` transforms.

0.0.1 Subscribed Topics

`laser_scan` (sensor_msgs/LaserScan)
Laser scans to process. (This topic can be remapped via the ~laser_scan_topic parameter)

`tf` (tf/tfMessage)
Transforms

0.0.2 Published Topics

`odom` (nav_msgs/Odometry)
Odometry estimations as a ROS topic. (This topic can be remapped via the ~odom_frame_id parameter)

`tf` (tf/tfMessage)
Publishes the transform from the \base_link (which can be remapped via the ~base_frame_id parameter) to \odom (which can be remapped via the ~odom_frame_id parameter).

0.0.3 Parameters

`~laser_scan_topic` (`string`, default: /laser_scan)
Topic name where lidar scans are being published.

`~base_frame_id` (`string`, default: /base_link)
TF frame name of the mobile robot base. A `tf` transform from the `laser_frame` to the `base_frame` should exist.

`~odom_frame_id` (`string`, default: /odom)
TF frame name for published odometry estimations. This same parameter is used to publish odometry as a topic.

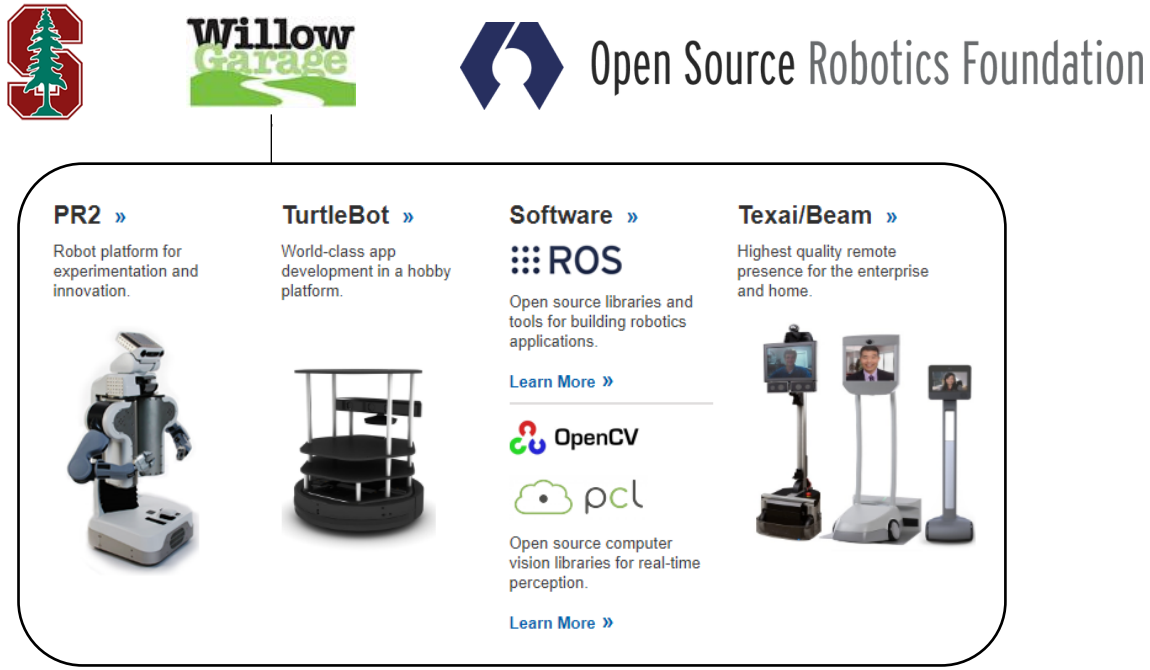
`~freq` (`double`, default: 10.0)
Odometry publication rate (Hz).

Wiki: rf2o_laser_odometry (última edición 2016-04-14 11:52:06 efectuada por JavierGMonroy)

Brought to you by: Open Robotics

3. ROS
EVOLUTION

- ROS started in 2007 in the Stanford University, then improved by Willow Garage Institute (2007-2013), and now under the Open Source Robotics Foundation.



3. ROS
EVOLUTION

Several releases (ROS distributions).

Distro	Release date	Poster	*Tuturtle*, turtle in tutorial	EOL date
ROS Noetic Ninjemys (Recommended)	May 23rd, 2020			May, 2025 (Focal EOL)
ROS Melodic Morenia	May 23rd, 2018			May, 2023 (Bionic EOL)
ROS Lunar Loggerhead	May 23rd, 2017			May, 2019
ROS Kinetic Kame	May 23rd, 2016			April, 2021 (Xenial EOL)
⋮				
ROS Box Turtle	March 2, 2010			--

Distro	Release date	Logo	EOL date
Humble Hawksbill	May 23rd, 2022		May 2027
Galactic Geochelone	May 23rd, 2021		December 9th, 2022
Foxy Fitzroy	June 5th, 2020		May 2023
Eloquent Elusor	November 22nd, 2019		November 2020
Dashing Diademata	May 31st, 2019		May 2021
Crystal Clemmys	December 14th, 2018		December 2019
Bouncy Bolson	July 2nd, 2018		July 2019
Ardent Apalone	December 8th, 2017		December 2018

3. ROS
RELEVANCE

In 2009, the paper introducing ROS "ROS: An Open-Source Robot Operating System" was presented at the IEEE International Conference on Robotics and Automation (ICRA). As of this month (Dec. 2024), has been cited 11,760 times.

ROS: an open-source Robot Operating System

Morgan Quigley*, Brian Gerkey†, Ken Conley†, Josh Faust†, Tully Foote‡, Jeremy Leibs‡, Eric Berger†, Rob Wheeler†, Andrew Ng*

*Computer Science Department, Stanford University, Stanford, CA
†Willow Garage, Menlo Park, CA
‡Computer Science Department, University of Southern California

Abstract— This paper gives an overview of ROS, an open-source robot operating system. ROS is not an operating system in the traditional sense of process management and scheduling; rather, it provides a structured communications layer above the host operating systems of a heterogenous compute cluster. In this paper, we discuss how ROS relates to existing robot software frameworks, and briefly overview some of the available application software which uses ROS.

I. INTRODUCTION

Writing software for robots is difficult, particularly as the scale and scope of robotics continues to grow. Different types of robots can have wildly varying hardware, making code reuse nontrivial. On top of this, the sheer size of the required code can be daunting, as it must contain a deep stack starting from driver-level software and continuing up through perception, abstract reasoning, and beyond. Since the required breadth of expertise is well beyond the capabilities of any single researcher, robotics software architectures must also support large-scale software integration efforts.

To meet these challenges, many robotics researchers, including ourselves, have previously created a wide variety of frameworks to manage complexity and facilitate rapid prototyping of software for experiments, resulting in the many robotic software systems currently used in academia and industry [1]. Each of these frameworks was designed for a particular purpose, perhaps in response to perceived weaknesses of other available frameworks, or to place emphasis on aspects which were seen as most important in the design process.

ROS, the framework described in this paper, is also the product of tradeoffs and prioritizations made during its design cycle. We believe its emphasis on large-scale integrative robotics research will be useful in a wide variety of situations as robotic systems grow ever more complex. In this paper, we discuss the design goals of ROS, how our implementation works towards them, and demonstrate how ROS handles several common use cases of robotics software development.

II. DESIGN GOALS

We do not claim that ROS is the best framework for all robotics software. In fact, we do not believe that such a framework exists; the field of robotics is far too broad for a single solution. ROS was designed to meet a specific set of challenges encountered when developing large-scale service robots as part of the STAIR project [2] at Stanford University[1] and the Personal Robots Program [3] at Willow Garage,[2] but the resulting architecture is far more general than the service-robot and mobile-manipulation domains.

Fig. 1. A typical ROS network configuration

The philosophical goals of ROS can be summarized as:

- Peer-to-peer
- Tools-based
- Multi-lingual
- Thin
- Free and Open-Source

To our knowledge, no existing framework has this same set of design criteria. In this section, we will elaborate these philosophies and shows how they influenced the design and implementation of ROS.

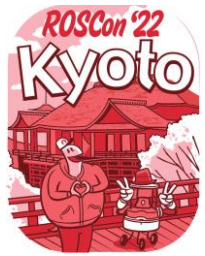
A. Peer-to-Peer

A system built using ROS consists of a number of processes, potentially on a number of different hosts, connected at runtime in a peer-to-peer topology. Although frameworks based on a central server (e.g., CARMEN [4]) can also realize the benefits of the multi-process and multi-host design, a central data server is problematic if the computers are connected in a heterogenous network.

For example, on the large service robots for which ROS was designed, there are typically several onboard computers connected via ethernet. This network segment is bridged via wireless LAN to high-power offboard machines that are running computation-intensive tasks such as computer vision or speech recognition (Figure 1). Running the central server either onboard or offboard would result in unnecessary

[1]http://stair.stanford.edu
[2]http://pr.willowgarage.com

It has a self-dedicated conference: **RosCon**



Adopted by relevant companies: BMW, ABB, 3M, Siemens, Boeing, etc. The list is huge! [take a look]

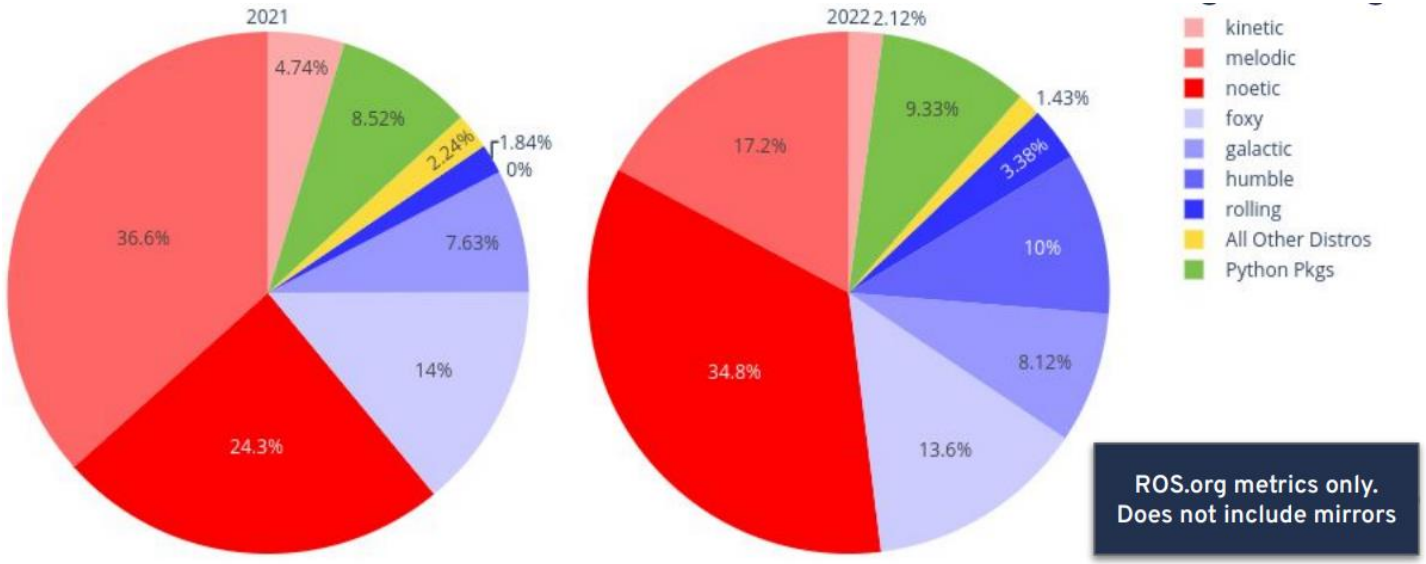
Recently created the ROS Industrial Consortium, developing ROS libraries (**ROS-I**) for industrial robots.

A new version is also available, **ROS 2**.

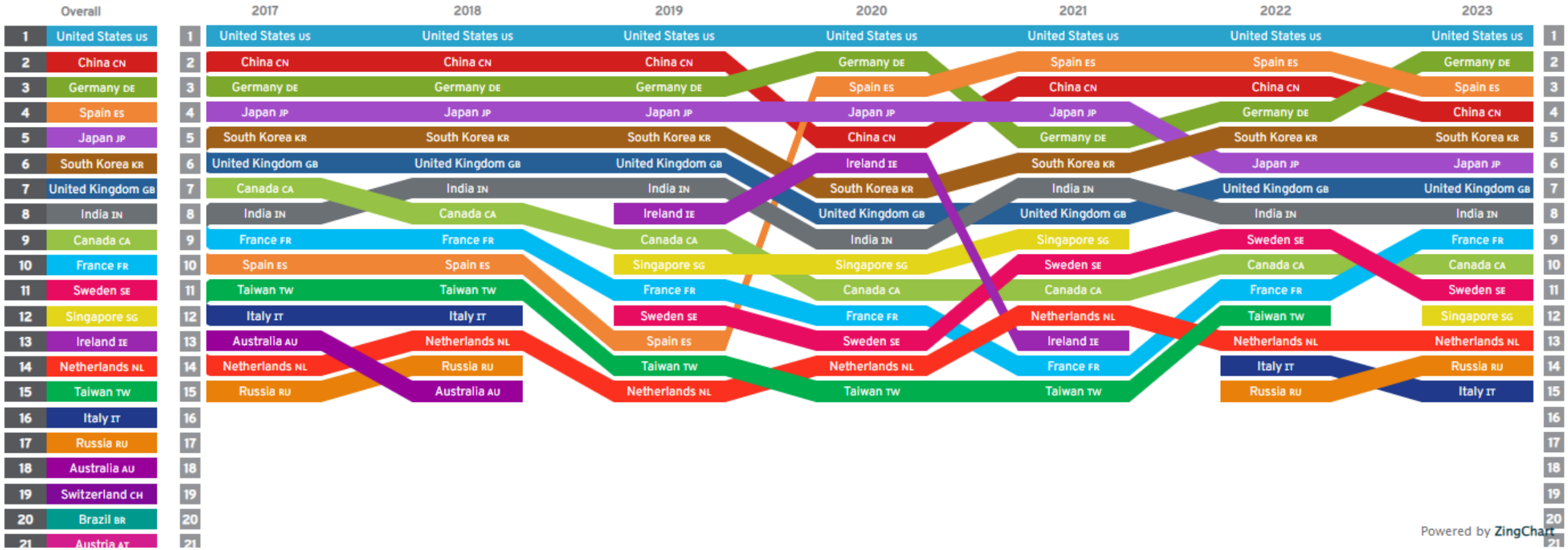
3. ROS
DISSEMINATION

- In 2022 about 591,3 million ROS packages were downloaded (212 million ROS 2).
- Strong community:
  - answers.ros.org (now moved to robotics.stackexchange.com): 67,427 questions (6,4% year increase) from which 43,494 are answered (15,7% increase).
  - discourse.ros.org ROS Community Discussion Forum (7.1K topics, 64,7K messages)

[Report here]



3. ROS
DISSEMINATION



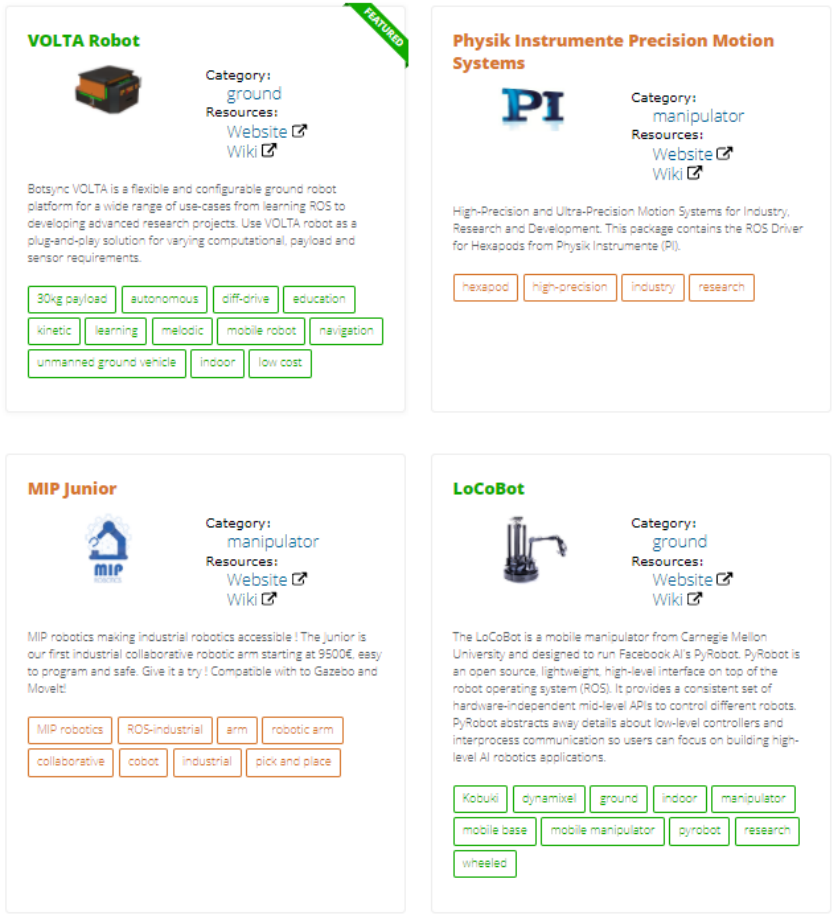
Top ROS-using countries based on packages.ros.org downloads.

[More metrics here]

3. ROS
DISSEMINATION

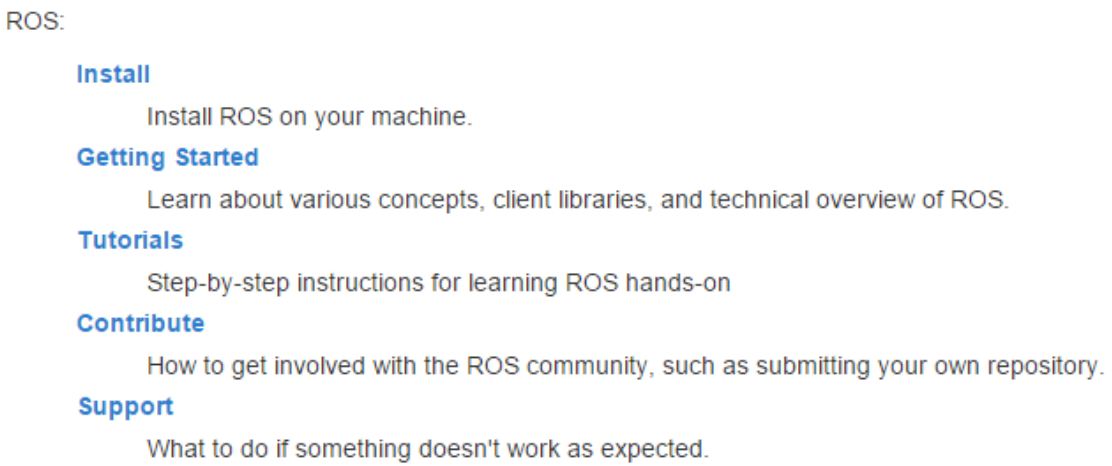
- Documented ROS robots:

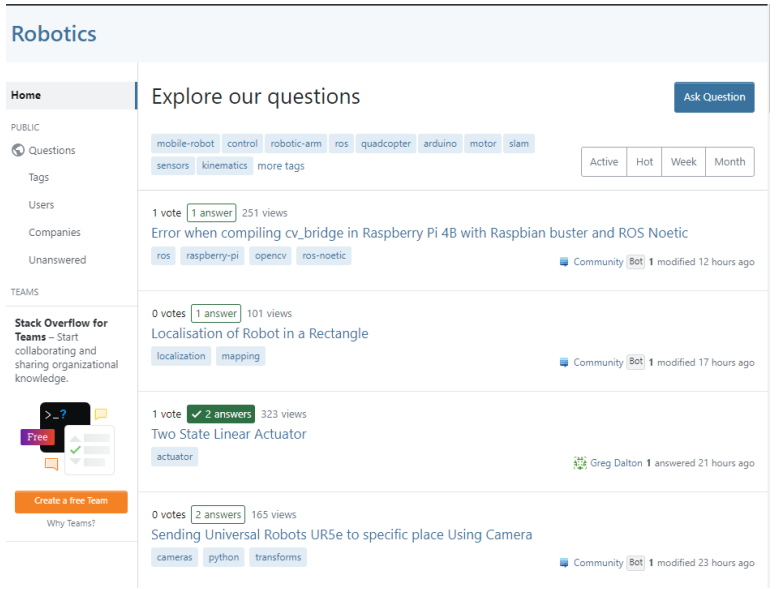
The number of different types of robots available to the community with ROS drivers. [link]



3. ROS
INSTALLATION AND HELP

- There are many installation guides on the www. Not covered here.
  - For example, detailed installation steps and more can be found at:
    - http://wiki.ros.org (ROS)
    - https://docs.ros.org/en/humble/Installation.html (ROS 2)
  - How to get help? robotics.stackexchange.com

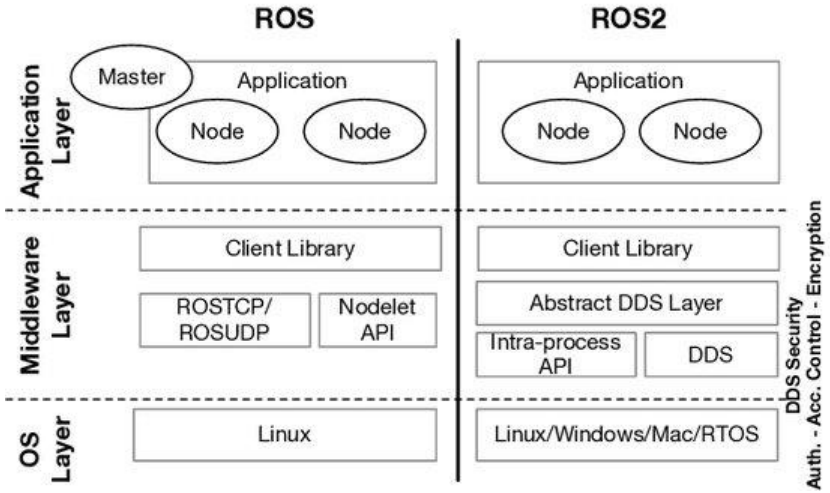




EXTRA. ROS2
ROS VS ROS 2

- Why ROS 2? Necessity of increasing security and real time features.
- Main differences:
  - Communications:
    - Communication network in ROS 1 followed a custom protocol.
    - ROS 2 uses a standard communication protocol based on DDS (Data Distribution Service). Different implementations exist.
    - This improves efficiency, security, scalability and Quality of Service (QoS)
  - No master:
    - Each node reports his presence to the rest in the network. Informs about his advertise and subscriptions.
    - Periodically repeats this process for new incoming nodes.
    - It also informs when it is shutdown.

Image from: Real-time Jitter Measurements under ROS2: the Inverted Pendulum case



- Also the programming languages supported: C++11 and Python2.7 vs C++14 or C++17 and Python 3.5.

4. ROS 2 – COMPONENTS AND COMMANDS

4. ROS 2 – COMPONENTS AND COMMANDS
PACKAGES

ROS 2 ECOSYSTEM

Design/Implementation phase

Packages
- Nodes

Navigation package
- Potential Fields

Map building package
- EKF SLAM
- Graph SLAM

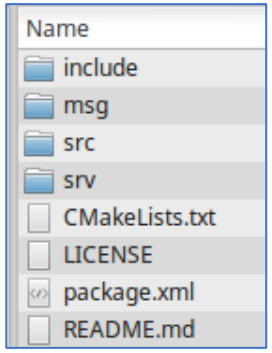
Perception package
- Landmark detection
- Landmark matching
- Laser readings

Execution/working phase

4. ROS 2 – COMPONENTS AND COMMANDS
PACKAGES

- **Packages**: All ROS 2 software is organized into packages. A ROS 2 package is a coherent collection of files, generally including both executables and supporting files, that serves a specific purpose.
- Packages can be created by using either CMake or Python, each one requiring their own configuration files but with one in common: package.xml (a manifest)

ROS package



Example of a typical CMake package

Packages-related commands:

$ ros2 pkg list

lists all the installed ROS 2 packages

$ ros2 pkg create

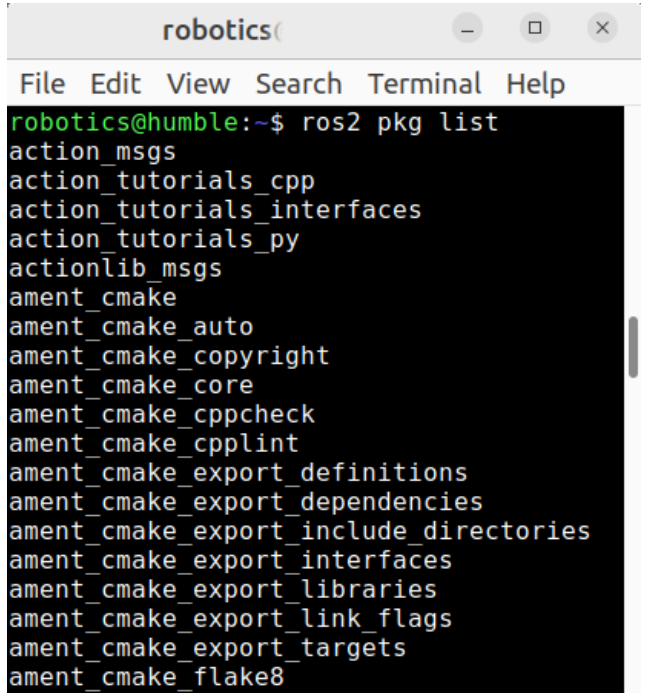
creates a package

$ ros2 pkg executables

lists the executables provided by a package

$ ros2 pkg prefix

returns the prefix path of a package



4. ROS 2 – COMPONENTS AND COMMANDS
PACKAGES

Example: package.xml

```
<?xml version="1.0"?>
<?xml-model href="http://download.ros.org/schema/package_format3.xsd" schematypens="http://www.w3.org/2001/XMLSchema"?>
<package format="3">
<package>
  <name>fake_battery</name>
  <version>0.0.1</version>
  <description>A fake battery package for simulation purposes</description>
  <maintainer email="jgmonroy@uma.es">Javier Monroy</maintainer>
  <license>GPLv3</license>

  <buildtool_depend>ament_cmake</buildtool_depend>

  <depend>rclcpp</depend>
  <depend>std_msgs</depend>

 <export>
   <build_type>ament_cmake</build_type>
 </export>
</package>
```

4. ROS 2 – COMPONENTS AND COMMANDS
NODES

ROS 2 ECOSYSTEM

Design/Implementation phase

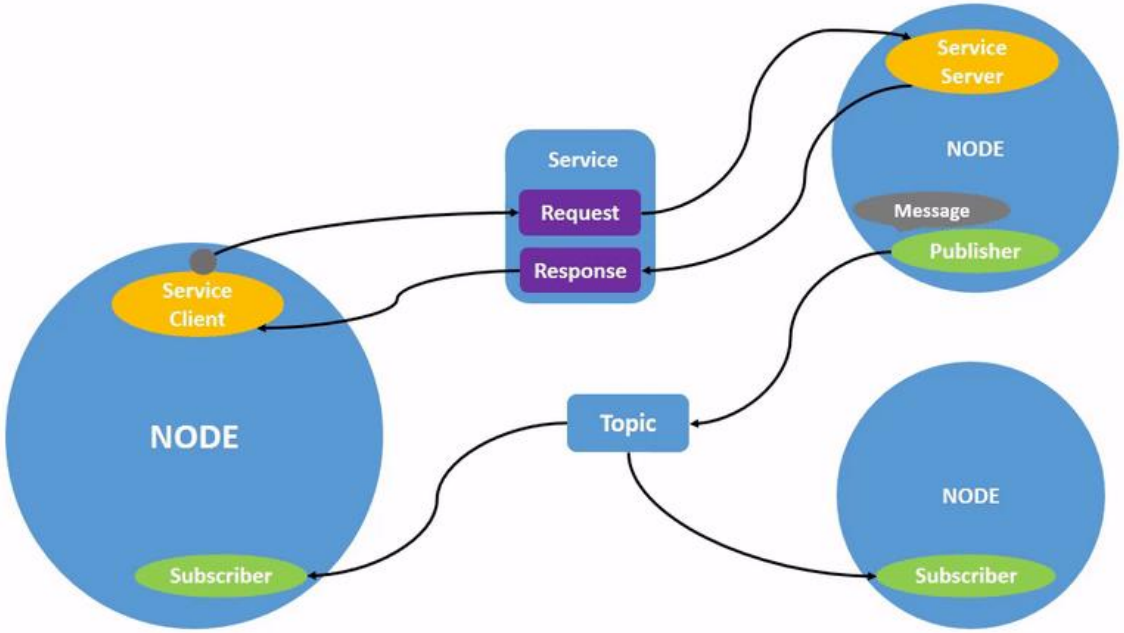
- Packages
  - Nodes
- Navigation package
  - Potential Fields
- Map building package
  - EKF SLAM
  - Graph SLAM
- Perception package
  - Landmark detection
  - Landmark matching
  - Laser readings

Execution/working phase

4. ROS 2 – COMPONENTS AND COMMANDS
NODES

- A robotic control architecture is comprised of many nodes working in concert.
- A **node** is a running instance of a ROS program (a single executable can contain one or more nodes).
- Performs computation and provide/consume data from other nodes via topics, services, actions or parameters.
- They should be responsible for a single purpose, e.g. publishing data from a laser, controlling the wheel motors, giving a localization, etc.
- Nodes are organized in packages (one or more multiple nodes can be organized within a package).

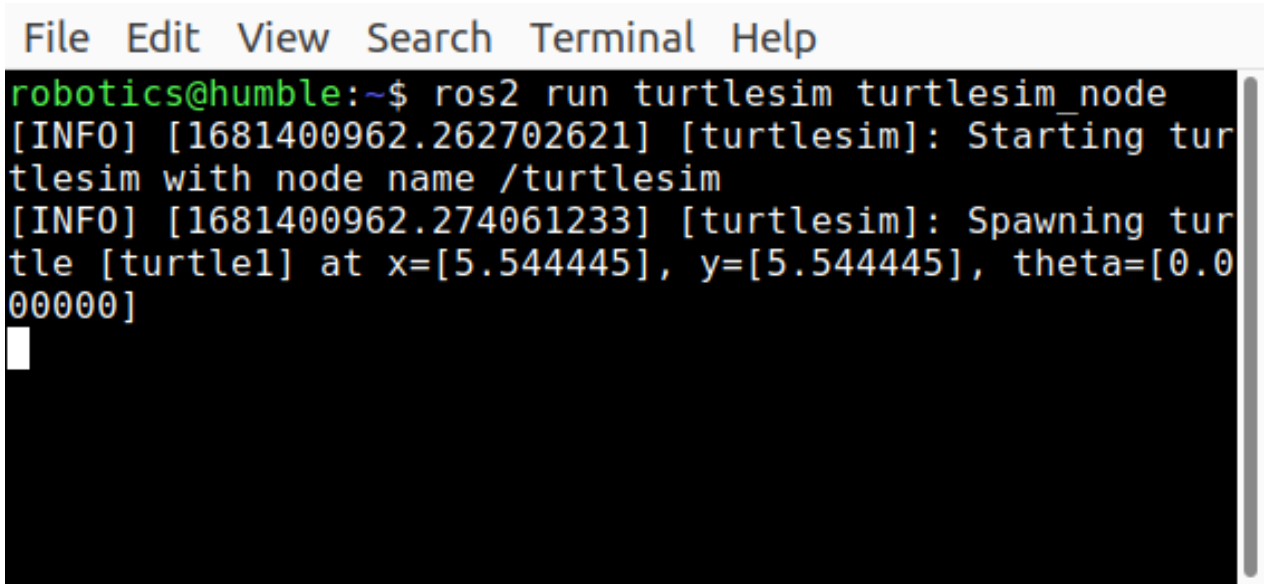
Taken from docs.ros.org

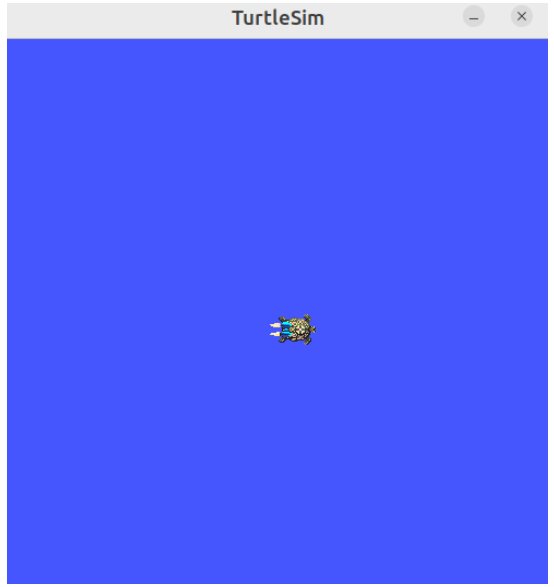


This is an example of the ROS 2 graph, a network of ROS 2 elements processing data together at the same time. Includes all executables and contections between tem.

4. ROS 2 – COMPONENTS AND COMMANDS
NODES

- **`ros2 run`** executes a node. The syntax is:
  - *`$ ros2 run <package> <executable>`*
- For instance, there is a test package called turtlesim which contains, among others, the turtlesim_node executable:





4. ROS 2 – COMPONENTS AND COMMANDS
NODES

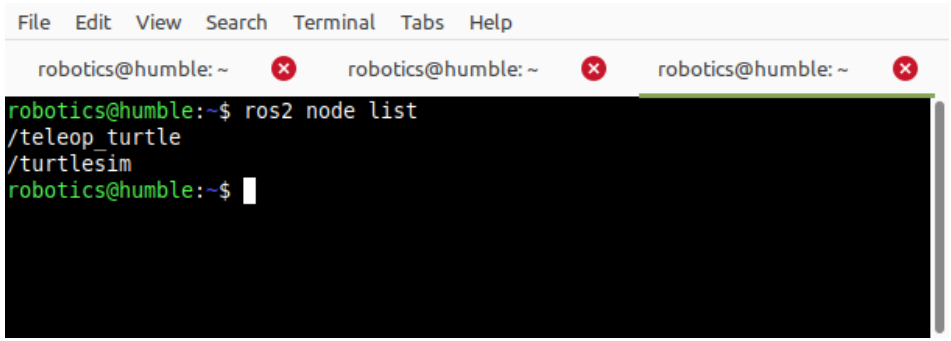
- **`ros2 node`** manages information about the running ROS 2 nodes.

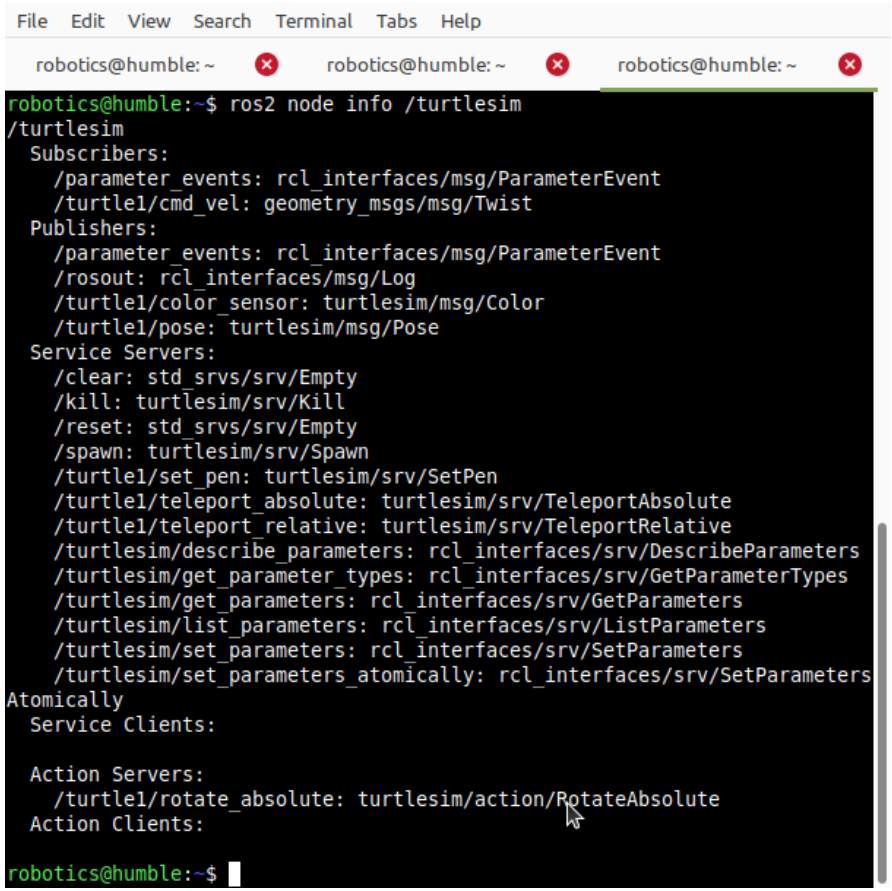
`$ ros2 node list`

lists running nodes

`$ ros2 node info <node_name>`

gives info about subscribers, publishers, services, and actions, that is the ROS graph connections that interact with that node





4. ROS 2 – COMPONENTS AND COMMANDS
NODES

- Another way to run a bunch of ROS 2 nodes: **launch files**
- Specifies which programs to run, where, whit which arguments, etc.
- They can be written in python, XML, or YAML.
- Typically created within a launch directory in your package.
- Two execution ways:
  - If within a package: `$ ros2 launch <package_name> <launch_file_name>`
  - If as *free souls*: `$ ros2 launch <launch_file_name>`

4. ROS 2 – COMPONENTS AND COMMANDS
NODES

launch files examples

xml

```xml
<launch>
  <node pkg="turtlesim" exec="turtlesim_node" name="sim" namespace="turtlesim1"/>
  <node pkg="turtlesim" exec="turtlesim_node" name="sim" namespace="turtlesim2"/>
  <node pkg="turtlesim" exec="mimic" name="mimic">
    <remap from="/input/pose" to="/turtlesim1/turtle1/pose"/>
    <remap from="/output/cmd_vel" to="/turtlesim2/turtle1/cmd_vel"/>
  </node>
</launch>
```

python

```python
from launch import LaunchDescription
from launch_ros.actions import Node

def generate_launch_description():
    return LaunchDescription([
        Node(
            package='turtlesim',
            namespace='turtlesim1',
            executable='turtlesim_node',
            name='sim'
        ),
        Node(
            package='turtlesim',
            namespace='turtlesim2',
            executable='turtlesim_node',
            name='sim'
        ),
        Node(
            package='turtlesim',
            executable='mimic',
            name='mimic',
            remappings=[
                ('/input/pose', '/turtlesim1/turtle1/pose'),
                ('/output/cmd_vel', '/turtlesim2/turtle1/cmd_vel'),
            ]
        )
    ])
```

4. ROS 2 – COMPONENTS AND COMMANDS
TOPICS

ROS 2 ECOSYSTEM

Design/Implementation phase

Packages
- Nodes

Navigation package
- Potential Fields

Map building package
- EKF SLAM
- Graph SLAM

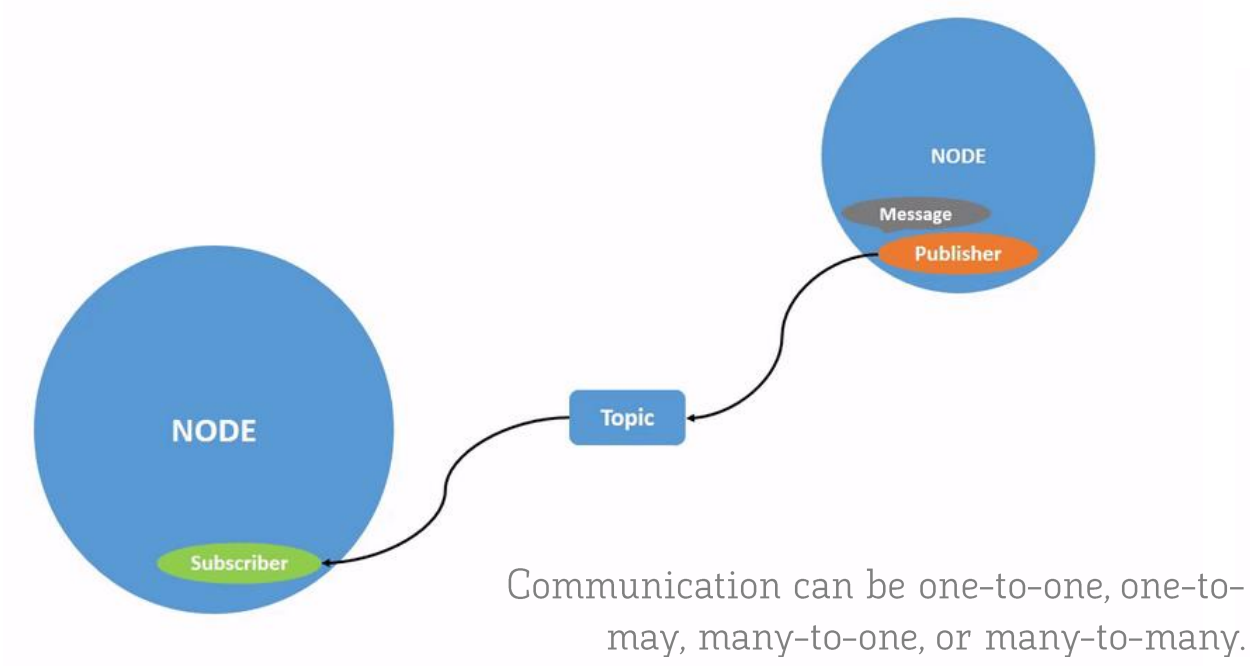
Perception package
- Landmark detection
- Landmark matching
- Laser readings

Execution/working phase

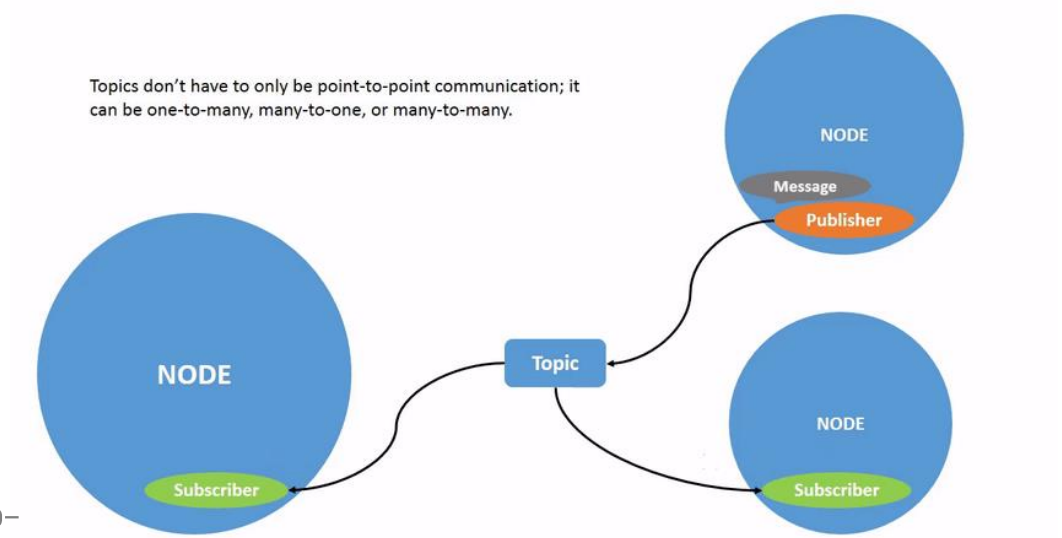
4. ROS 2 – COMPONENTS AND COMMANDS
TOPICS

- The primary mechanism that ROS 2 nodes use to communicate is to send messages.
- Messages in ROS 2 are organized into named topics.
- Topics act as a bus for nodes to exchange messages.

- A node that wants to share information will publish messages on the appropriate topic or topics.
- A node that wants to receive information will subscribe to the topic or topics that it's interested in.



Communication can be one-to-one, one-to-may, many-to-one, or many-to-many.



4. ROS 2 – COMPONENTS AND COMMANDS
TOPICS

- **`ros2 topic`** manages information about the published ROS topics.

`$ ros2 topic list`

lists active topics

`$ ros2 topic info <topic_name>`

gives info about an active topic

`$ ros2 topic echo <topic_name>`

dump messages published to the terminal

`$ ros2 topic pub -r rate_hz <topic_name> <message_type> <message_content>`

publishes a msg over a given topic

`$ ros2 interface show <topic-type>`

shows the data parameters of a given topic

4. ROS 2 – COMPONENTS AND COMMANDS
TOPICS

- Some examples:

```
File  Edit  View  Search  Terminal  Tabs  Help
turtles...   teleop...   roboti...
robotics@humble:~$ ros2 topic list
/parameter_events
/rosout
/turtle1/cmd_vel
/turtle1/color_sensor
/turtle1/pose
robotics@humble:~$
```

```
File  Edit  View  Search  Terminal  Tabs  Help
turtlesim   teleoperation   robotics@h...
robotics@humble:~$ ros2 topic info /turtle1/cmd_vel
Type: geometry_msgs/msg/Twist
Publisher count: 1
Subscription count: 1
robotics@humble:~$
```

```
File  Edit  View  Search  Terminal  Tabs  Help
turtlesim   teleoperation   robotics@h...
robotics@humble:~$ ros2 topic list -t
/parameter_events [rcl_interfaces/msg/ParameterEvent]
/rosout [rcl_interfaces/msg/Log]
/turtle1/cmd_vel [geometry_msgs/msg/Twist]
/turtle1/color_sensor [turtlesim/msg/Color]
/turtle1/pose [turtlesim/msg/Pose]
robotics@humble:~$
```

```
File  Edit  View  Search  Terminal  Tabs  Help
turtlesim   teleoperation   robotics@h...
robotics@humble:~$ ros2 topic type /turtle1/cmd_vel
geometry_msgs/msg/Twist
robotics@humble:~$
```

A topic is defined over a message of a certain type, which can be further checked with **`ros2 interface show <msg-name>`**:

```
File  Edit  View  Search  Terminal  Tabs  Help
turtlesim   teleoperation   robotics@hum...
robotics@humble:~$ ros2 interface show geometry_msgs/msg/Twist
# This expresses velocity in free space broken into its linear
and angular parts.

Vector3  linear
        float64 x
        float64 y
        float64 z
Vector3  angular
        float64 x
        float64 y
        float64 z
robotics@humble:~$
```

/geometry_msgs/msg/Twist (in the geometry_msgs package there is a msg called Twist) contains two fields: linear and angular which are Vector3 typed.

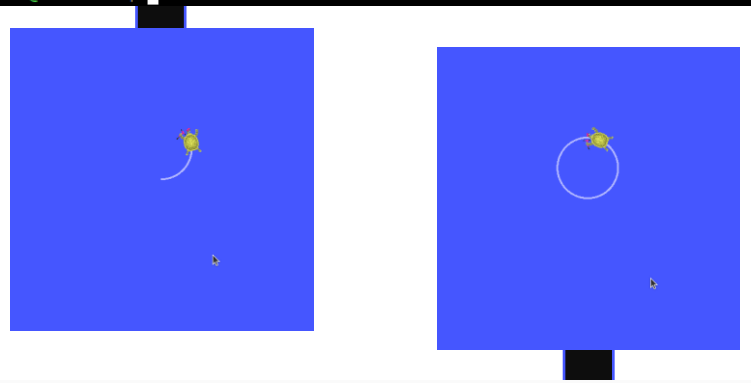
4. ROS 2 – COMPONENTS AND COMMANDS
TOPICS

- Example of publishing on a topic from the command line:

Publishes a latched message on the `cmd_vel` topic.

```
File  Edit  View  Search  Terminal  Tabs  Help
robotics@humble: ~   robotics@humble: ~   robotics@humble: ~
robotics@humble:~$ ros2 topic pub --once /turtle1/cmd_vel geometry_msgs/msg/Twist
"{linear: {x: 2.0, y: 0.0, z: 0.0}, angular: {x: 0.0, y: 0.0, z: 1.8}}"
publisher: beginning loop
publishing #1: geometry_msgs.msg.Twist(linear=geometry_msgs.msg.Vector3(x=2.0, y=0
.0, z=0.0), angular=geometry_msgs.msg.Vector3(x=0.0, y=0.0, z=1.8))

robotics@humble:~$
```



Publishes a message on the `cmd_vel` topic every second.

```
File  Edit  View  Search  Terminal  Tabs  Help
robotics@humble: ~   robotics@humble: ~   robotics@humble: ~
robotics@humble:~$ ros2 topic pub --rate 1 /turtle1/cmd_vel geometry_msgs/msg/Twis
t "{linear: {x: 2.0, y: 0.0, z: 0.0}, angular: {x: 0.0, y: 0.0, z: 1.8}}"
publisher: beginning loop
publishing #1: geometry_msgs.msg.Twist(linear=geometry_msgs.msg.Vector3(x=2.0, y=0
.0, z=0.0), angular=geometry_msgs.msg.Vector3(x=0.0, y=0.0, z=1.8))

publishing #2: geometry_msgs.msg.Twist(linear=geometry_msgs.msg.Vector3(x=2.0, y=0
```

```
File  Edit  View  Search  Terminal  Tabs  Help
robotics...  robotics...  robotics...  robotics...
robotics@humble:~$ ros2 topic hz /turtle1/cmd_vel
WARNING: topic [/turtle1/cmd_vel] does not appear to be publis
hed yet
average rate: 1.000
        min: 1.000s max: 1.001s std dev: 0.00069s window: 3
average rate: 1.000
        min: 0.999s max: 1.001s std dev: 0.00083s window: 5
average rate: 1.000
        min: 0.999s max: 1.001s std dev: 0.00086s window: 7
average rate: 1.000
        min: 0.999s max: 1.001s std dev: 0.00085s window: 9
average rate: 1.000
        min: 0.999s max: 1.001s std dev: 0.00087s window: 11
```

View the rate at which data is published

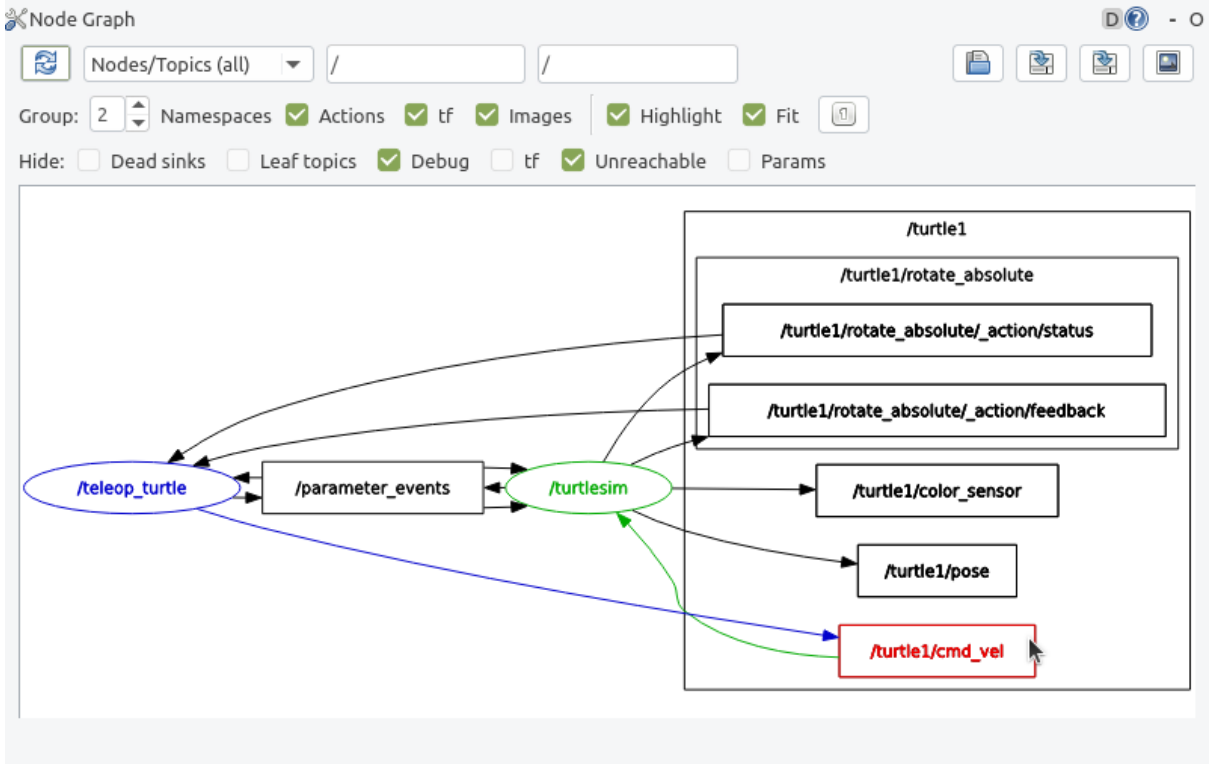
```
File  Edit  View  Search  Terminal  Tabs  Help
robo...  robo...  robo...  robo...
robotics@humble:~$ ros2 topic echo /turtle1/pose
x: 6.17805814743042
y: 5.754004955291748
theta: 0.6100262999534607
linear_velocity: 2.0
angular_velocity: 1.7999999523162842
---
x: 6.203747749328613
y: 5.773085117340088
theta: 0.638826310634613
linear_velocity: 2.0
angular_velocity: 1.7999999523162842
---
```

Echo the message values.

4. ROS 2 – COMPONENTS AND COMMANDS
TOPICS

- `rqt_graph` is a ROS node that graphically shows the topics subscriptions of the running nodes. Run it as a normal node:

  `$ ros2 run rqt_graph rqt_graph` or just *`$ rqt_graph`*



ROS 2 ECOSYSTEM

Design/Implementation phase

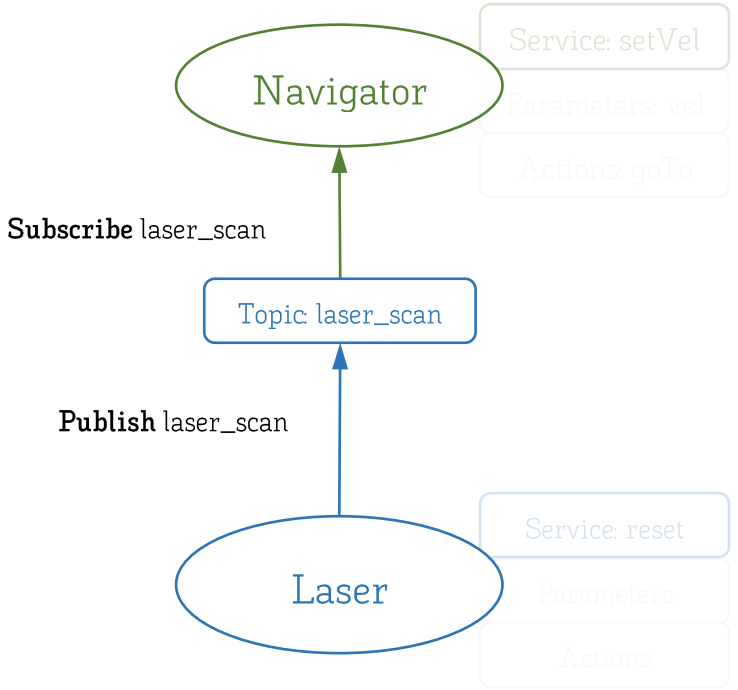
Packages
- Nodes

Navigation package
- Potential Fields

Map building package
- EKF SLAM
- Graph SLAM

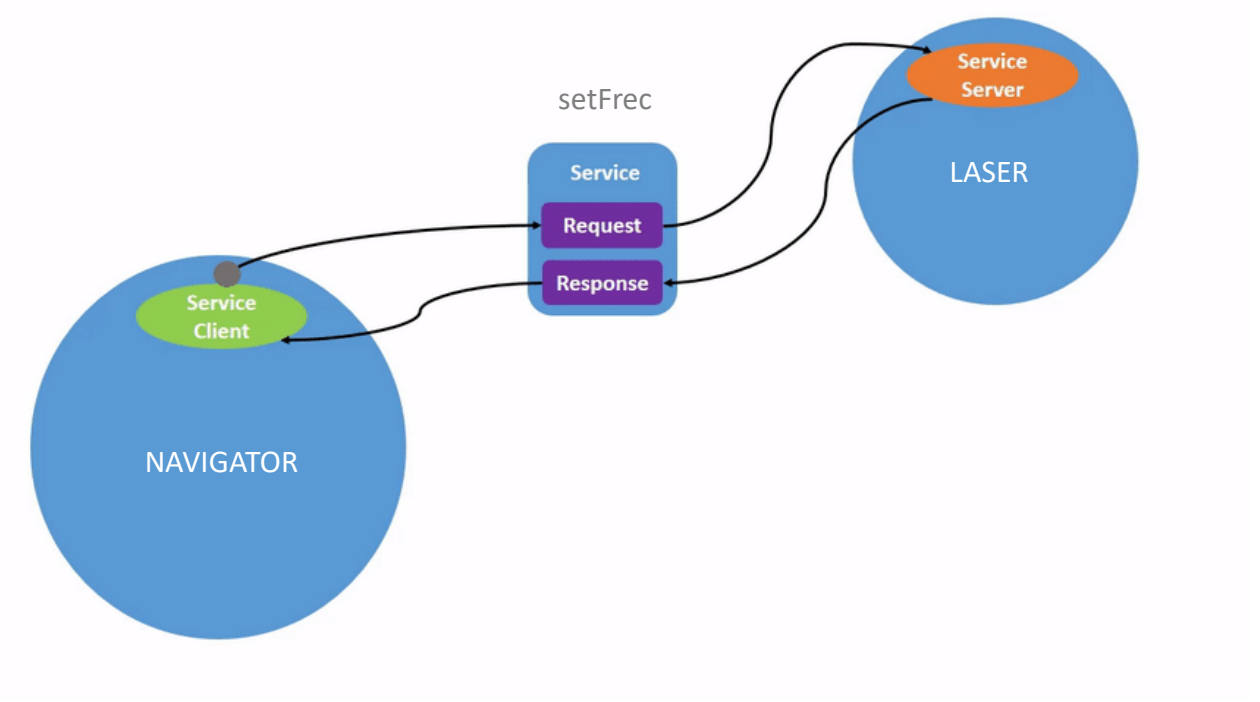
Perception package
- Landmark detection
- Landmark matching
- Laser readings

Execution/working phase



4. ROS 2 – COMPONENTS AND COMMANDS
SERVICES

- Services allow a node (the client of the service) to request a particular operation to another node (the server of the service).
- The server answers with a structured response.
- Call-and-response model vs publisher-subscriber model of topics.



4. ROS 2 – COMPONENTS AND COMMANDS
SERVICES

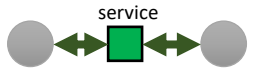
- `ros2 service` manages information about the active services.

$ ros2 service list
lists avaiable services

$ ros2 service type <service>
gives the type of a service

$ ros2 service find <service>
outputs a list of services of a given type

$ ros2 service call <service>
calls (executes) a service



4. ROS 2 – COMPONENTS AND COMMANDS
SERVICES

- Some examples:

Listing the available services

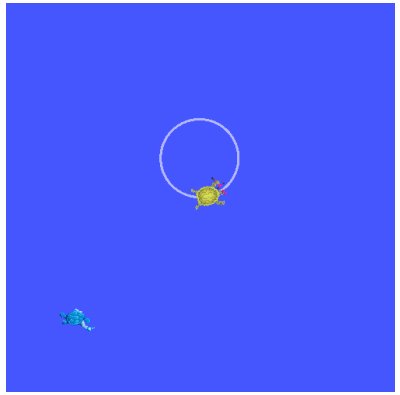
```
robotics@humble:~$ ros2 service list
/clear
/kill
/reset
/spawn
/turtle1/set_pen
/turtle1/teleport_absolute
/turtle1/teleport_relative
/turtlesim/describe_parameters
/turtlesim/get_parameter_types
/turtlesim/get_parameters
/turtlesim/list_parameters
/turtlesim/set_parameters
/turtlesim/set_parameters_atomically
robotics@humble:~$
```

Calling a service. It answers with a string value, in this case, the turtle name.

```
robotics@humble:~$ ros2 service call /spawn turtlesim/srv/Spawn "{x: 2, y: 2, theta: 0.2, name: ''}"
requester: making request: turtlesim.srv.Spawn_Request(x=2.0, y=2.0, theta=0.2, name='')

response:
turtlesim.srv.Spawn_Response(name='turtle2')

robotics@humble:~$
```



The structure of a service is defined over a message of a certain type, codified into a `.srv` file, which can be further checked with `ros2 interface`

```
robotics@humble:~$ ros2 service type /spawn
turtlesim/srv/Spawn
robotics@humble:~$ ros2 interface show turtlesim/srv/Spawn
float32 x
float32 y
float32 theta
string name # Optional.  A unique name will be created and returned if this is empty
---
string name
robotics@humble:~$
```

Request

Response

ROS 2 ECOSYSTEM

Design/Implementation phase

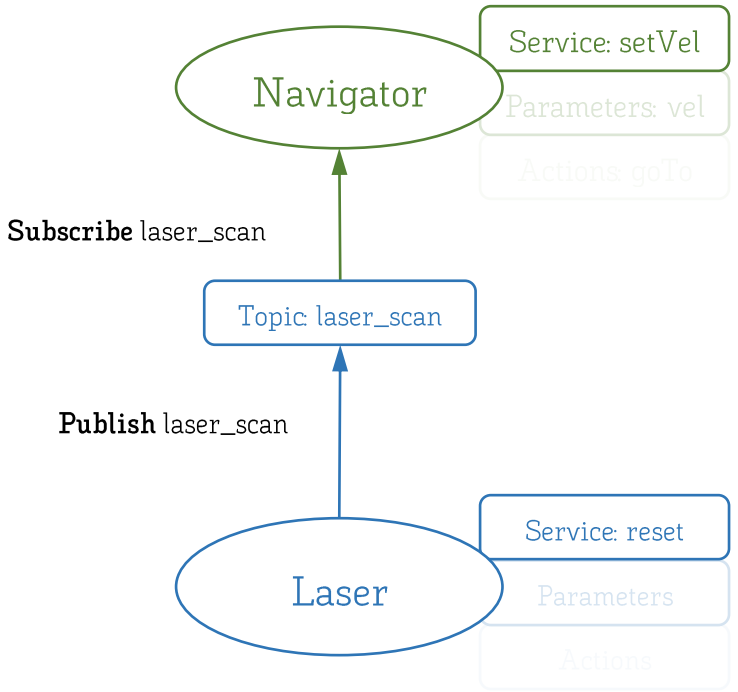
Packages
- Nodes

Navigation package
- Potential Fields

Map building package
- EKF SLAM
- Graph SLAM

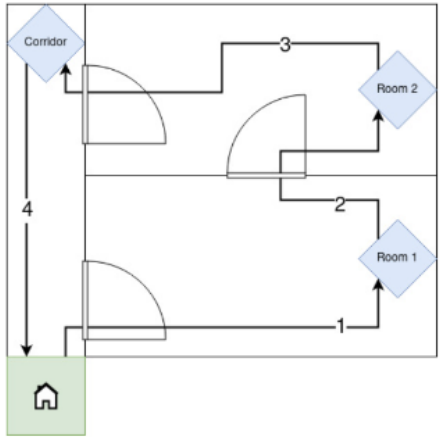
Perception package
- Landmark detection
- Landmark matching
- Laser readings

Execution/working phase



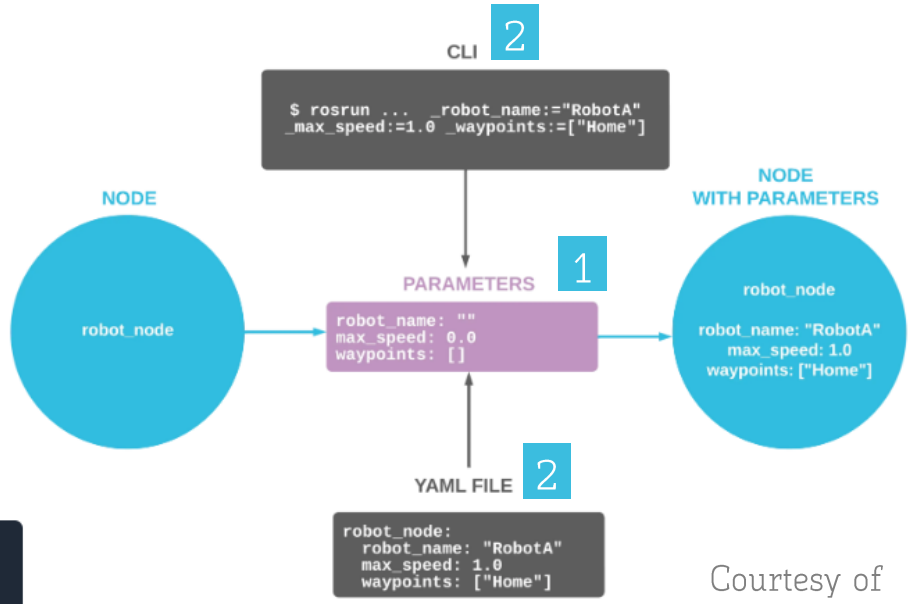
4. ROS 2 – COMPONENTS AND COMMANDS
PARAMETERS

- **Parameter:** a configuration value of a node used at startup and during runtime.
- They can be integers, floats, booleans, strings, and lists in <type,value,description> format.
- In ROS 2 each node maintains its own parameters (vs. parameters server in ROS).




```
/robot_node:
ros__parameters:
robot_name: "RobotA"
max_speed: 1.4
waypoints: ["Home", "Corridor", "Home"]
```

```
/robot_node:
  ros__parameters:
    robot_name: "RobotB"
    max_speed: 5.6
    waypoints: ["Home", "Room 1", "Room 2", "Corridor", "Home"]
```



Courtesy of
José L. Millán

4. ROS 2 – COMPONENTS AND COMMANDS
PARAMETERS

- **`ros2 param`** manages information about the parameters of the active nodes.

`$ ros2 param list`

lists active parameters

`$ ros2 param get <parameter>`

gets the value of a given parameter

`$ ros2 param set <parameter> <new_value>`

sets a new value to a given parameter

`$ ros2 param describe <parameter>`

gets descriptive information about parameter

`$ ros2 param dump <file> <namespace>`

saves parameters from a namespace to a file.

`$ ros2 param load <file> <namespace>`

loads parameters from file

ROS 2 ECOSYSTEM

Design/Implementation phase

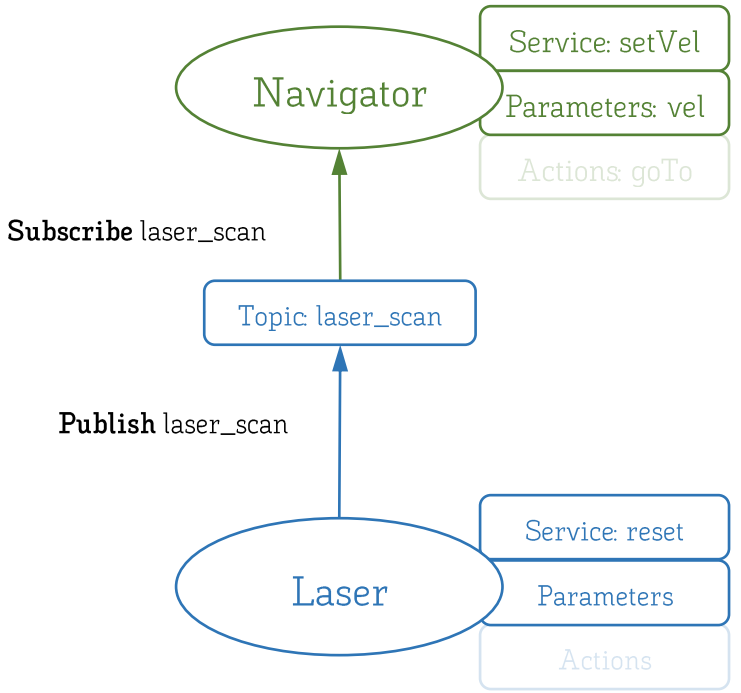
Packages
- Nodes

Navigation package
- Potential Fields

Map building package
- EKF SLAM
- Graph SLAM

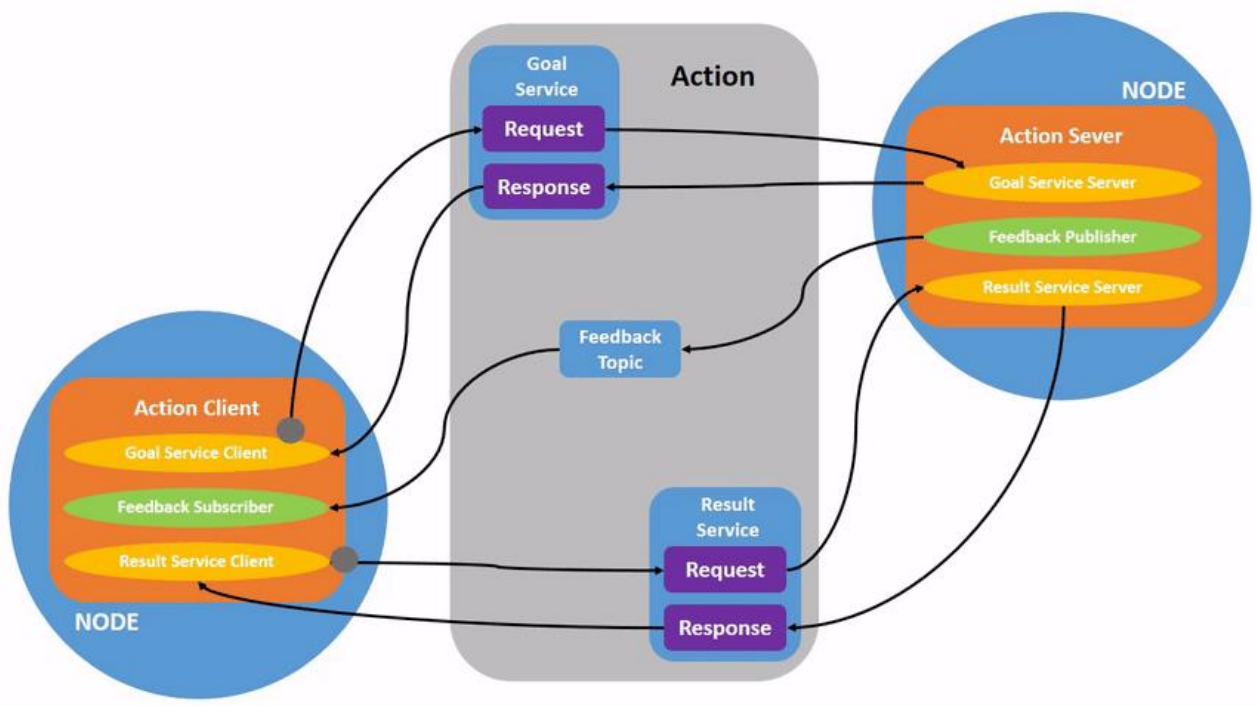
Perception package
- Landmark detection
- Landmark matching
- Laser readings

Execution/working phase



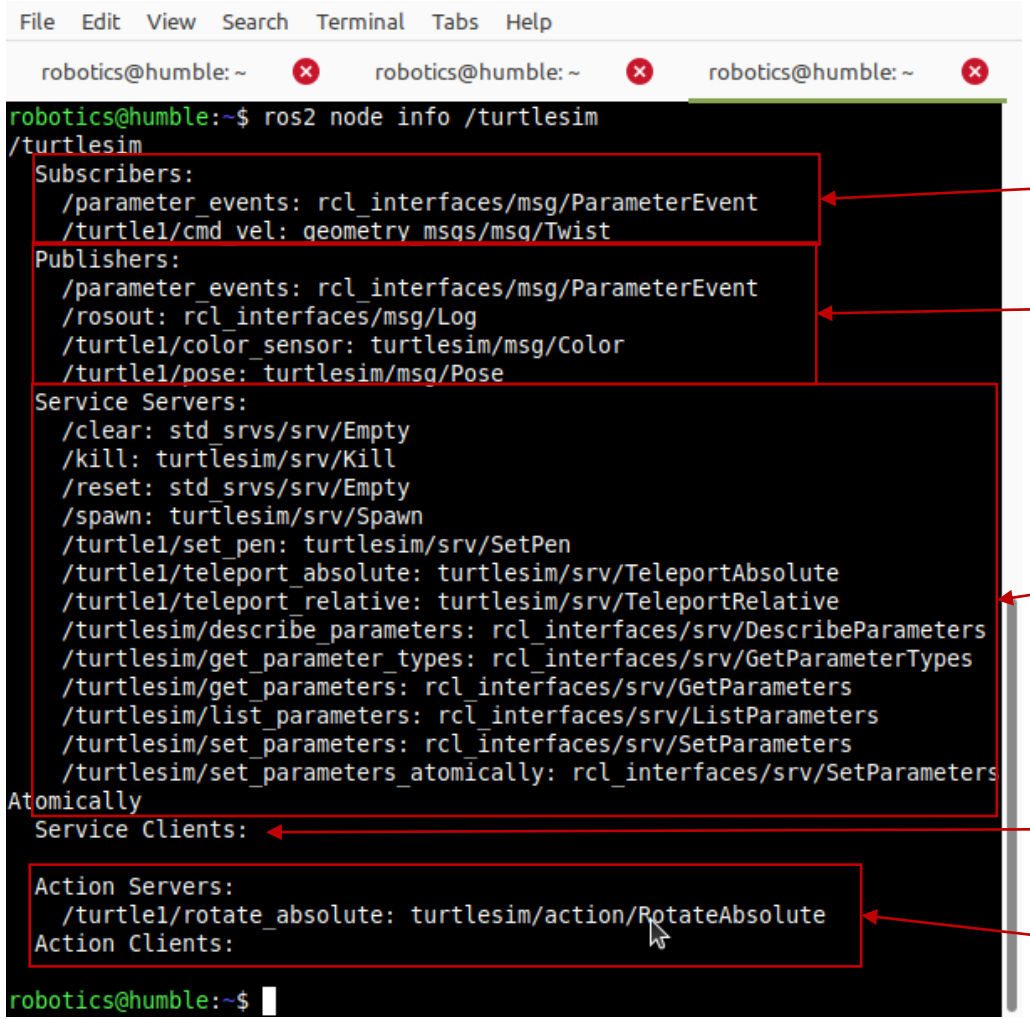
4. ROS 2 – COMPONENTS AND COMMANDS
ACTIONS

- The last communication type in ROS 2, intended for **long running tasks**.
- Can **cancel** the request during execution
- Provide periodic feedback about how the request is being processing.
- Follow a Server-Client scheme.



4. ROS 2 – COMPONENTS AND COMMANDS
FINAL REMARKS

Looking back, the *$ ros2 node info* command provides useful info.



Subscribers: Topics on which this node is listening

Publishers: Topics on which this node is sending data

Service Servers: Services provided by this node

Service Clients: Services used by this node. In this case none

Actions Servers/Clients: Non-blocking services

4. ROS 2 – COMPONENTS AND COMMANDS
FINAL REMARKS

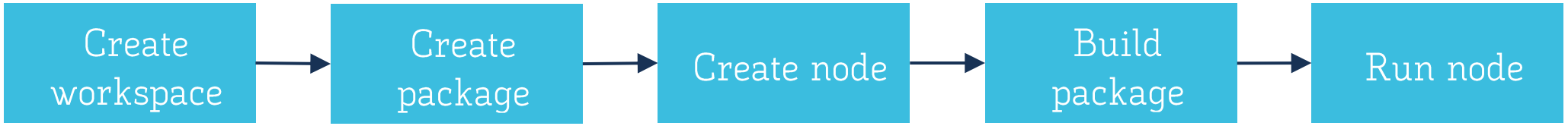
- We have looked at:
  - `ros2 topic`
  - `ros2 service`
  - `ros2 action`
  - `ros2 interface show <message>|<service>|<action>`
- Be clear on this:

	Topics	Services	Actions
Active things	`ros2 topic`	`ros2 service`	`ros2 action`
Data types (files)	`messages (.msg)`	`services (.srv)`	`actions (act.)`

5. ROS 2 – DEVELOPING SOFTWARE

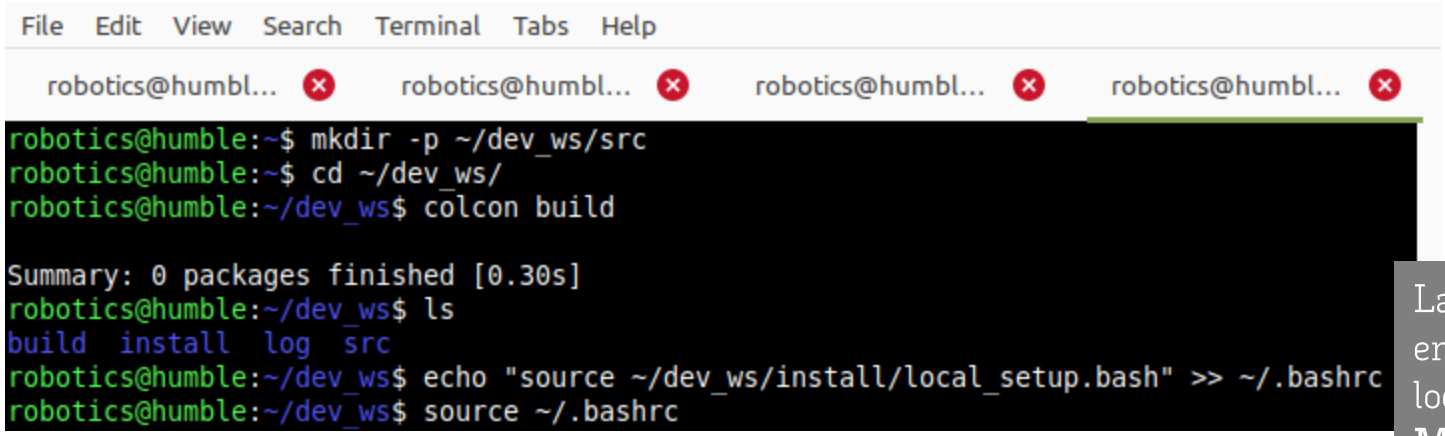
5. ROS 2 – DEVELOPING SOFTWARE
COLCON

- When you become a ROS developer, you aim to implement new nodes, which are organized into packages.
- colcon: official build tool of ROS based on Cmake and python. It is in charge of:
  - managing dependences among nodes,
  - organizing and making packages accessible, and
  - generating executables from source code.
- Workflow (from workspace to node):



5. ROS 2 – DEVELOPING SOFTWARE
COLCON WORKSPACE

- A **workspace** is a directory where colcon operates to build packages, with the following directories:
  - src: place to locate the source code of ROS 2 packages.
  - build: intermediate files are stored here.
  - install: contains the installation of each package.
  - log: contains diverse login information.
- The following commands create a workspace called `dev_ws` and its `/src` directory, move to it, calls to colcon build (this creates and initial set of directories and files), and list the resulting directories. With this we get a workspace ready to work with!



Last two lines add useful ROS 2 environment variables (e.g. where to look for installed executables) to bashrc. **MUST BE DONE ONLY ONCE!**

5. ROS 2 – DEVELOPING SOFTWARE
CREATING YOUR OWN PACKAGE

Run it only inside the `src` folder!!

- ros2 pkg create command, with **CMake:**

More convenient when developing C++ nodes

```
ros2 pkg create --build-type ament_cmake
<package_name>
```

```
File  Edit  View  Search  Terminal  Help
robotics@humble:~/ros2_ws/src$ ros2 pkg create --build-type
ament_cmake --node-name my_node my_package_cmake
going to create a new package
package name: my_package_cmake
destination directory: /home/robotics/ros2_ws/src
package format: 3
version: 0.0.0
description: TODO: Package description
maintainer: ['robotics <humble@example.com>']
licenses: ['TODO: License declaration']
build type: ament_cmake
dependencies: []
node_name: my_node
creating folder ./my_package_cmake
creating ./my_package_cmake/package.xml
creating source and include folder
creating folder ./my_package_cmake/src
creating folder ./my_package_cmake/include/my_package_cmake
creating ./my_package_cmake/CMakeLists.txt
creating ./my_package_cmake/src/my_node.cpp

[WARNING]: Unknown license 'TODO: License declaration'.  Thi
s has been set in the package.xml, but no LICENSE file has b
een created.
It is recommended to use one of the ament license identitife
rs:
Apache-2.0
BSL-1.0
```

This creates a directory with the same name and some initial content:

```
File  Edit  View  Search  Terminal  Help
robotics@humble:~/ros2_ws/src/my_package_cmake$ ls
CMakeLists.txt  include  package.xml  src
robotics@humble:~/ros2_ws/src/my_package_cmake$
```

In this case, as we called ros2 pkg create with the *–node–name* option, a node called my_node with a simple Hello World executable is created.

5. ROS 2 – DEVELOPING SOFTWARE
CREATING YOUR OWN PACKAGE

Run it only inside the `src` folder!!

- ros2 pkg create command, with **Python:**

More convenient when developing python nodes

```
ros2 pkg create --build-type ament_python
<package_name>
```

```
File  Edit  View  Search  Terminal  Help
Either remove the directory or choose a different destinatio
n directory or package name
robotics@humble:~/ros2_ws/src$ ros2 pkg create --build-type
ament_python --node-name my_node my_package_python
going to create a new package
package name: my_package_python
destination directory: /home/robotics/ros2_ws/src
package format: 3
version: 0.0.0
description: TODO: Package description
maintainer: ['robotics <humble@example.com>']
licenses: ['TODO: License declaration']
build type: ament_python
dependencies: []
node_name: my_node
creating folder ./my_package_python
creating ./my_package_python/package.xml
creating source folder
creating folder ./my_package_python/my_package_python
creating ./my_package_python/setup.py
creating ./my_package_python/setup.cfg
creating folder ./my_package_python/resource
creating ./my_package_python/resource/my_package_python
creating ./my_package_python/my_package_python/__init__.py
creating folder ./my_package_python/test
creating ./my_package_python/test/test_copyright.py
creating ./my_package_python/test/test_flake8.py
creating ./my_package_python/test/test_pep257.py
creating ./my_package_python/my_package_python/my_node.py
```

This creates a directory with the same name and some initial content:

```
File  Edit  View  Search  Terminal  Help
robotics@humble:~/ros2_ws/src/my_package_python$ ls
my_package_python  resource    setup.py
package.xml        setup.cfg   test
robotics@humble:~/ros2_ws/src/my_package_python$
```

In this case, as we called ros2 pkg create with the *–node–name* option, a node called my_node with a simple Hello World executable is created.

5. ROS 2 – DEVELOPING SOFTWARE
CREATING YOUR OWN PACKAGE

- Typical structure of the workspace:

```
workspace_folder/
    src/
      package_1/
          CMakeLists.txt       -- CMakeLists.txt file for package_1
          package.xml          -- Package manifest for package_1

      package_2/
          setup.py             -- Installation instructions for package_2
          package.xml          -- Package manifest for package_2
          resource/package_2
      ...
      package_n/
          CMakeLists.txt       -- CMakeLists.txt file for package_1
          package.xml          -- Package manifest for package_n
```

Remind:
- A single workspace can contain as many packages as you want.
- A package can have different build types (e.g. CMake or Python).
- No nested packages are allowed.

Remind:
- To build a package, just go to the worksapce root directory and call colcon with `colcon build`.
- To build only package_1: `coclon build --packages-select package_1`

5. ROS 2 – DEVELOPING SOFTWARE
IMPLEMENTING YOUR OWN NODE (CMAKE)

1. Create a new source file in the src folder of the package. For example, `my_node.cpp`
2. Modify the CMakeList.txt at `/src/my_package_cmake/` to include a reference to a such file. This tells the ROS 2 system we want to have a new executable (node).

```
 8 # find dependencies
 9 find_package(ament_cmake REQUIRED)
10 find_package(rclcpp REQUIRED)
11 find_package(std_msgs REQUIRED)
12
13 add_executable(my_node src/my_node.cpp)
14 ament_target_dependencies(my_node rclcpp std_msgs)
15
16 install(TARGETS my_node
17   DESTINATION lib/${PROJECT_NAME})
```

3. Develop the node itself. In our example, implement in `my_node.cpp` the desiserd functionality. You can use a simple text editor or an IDE (e.g. Visual Studio Code).
4. Compile the source code and generate the node executable.

```
$ ~/ros2_ws
$ colcon build
```

5. Run your node! `ros2 run my_package_cmake my_node`

Depending on your configuration, you may previously need to go to your workspace root directory and source the setup files:
`. install/setup.bash`

5. ROS 2 – DEVELOPING SOFTWARE
IMPLEMENTING YOUR OWN NODE (PYTHON)

1. Create a new python file in the my_package_python folder of the package. For example, `my_node.py`
2. Modify setup.py at `/src/my_package_python/` by updating the maintainer, email, description, and license (put the same as in package.xml) and add a new entry telling ROS 2 about the existence of your node:

```
16      maintainer='robotics',
17      maintainer_email='humble@example.com',
18      description='TODO: Package description',
19      license='TODO: License declaration',
20      tests_require=['pytest'],
21      entry_points={
22          'console_scripts': [
23              'my_node = my_package_python.my_node:main'
24          ],
25      },
```

3. Develop the node itself. In our example, implement in `my_node.py` the desired functionality. You can use a simple text editor or an IDE (e.g. Visual Studio Code).
4. Build the new package/node::

```
$ ~/ros2_ws
$ colcon build
```

5. Run your node! `$ ros2 run my_package_python my_node`

Depending on your configuration, you may previously need to go to your workspace root directory and source the setup files:
`$ source install/setup.bash`

REFERENCES
WHERE TO FIND MORE KNOWLEDGE

[1] ROS 2 Documentation: https://docs.ros.org/en/humble/index.html

[2] Quigley, Morgan, et al. "ROS: an open-source Robot Operating System." *ICRA workshop on open source software*. Vol. 3. No. 3.2. 2009.

[3] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, W. Woodall, "Robot Operating System 2: Design, architecture, and uses in the wild," Science Robotics vol. 7, May 2022.

SEE YOU IN THE NEXT LECTURE!