Robotics Lab
Master's Course in Automation Engineering and Robotics
Academic Year 2024/2025

Prof. Mario Selvaggio
E-mail: mario.selvaggio@unina.it
Website: http://wpage.unina.it/mario.selvaggio/index.html

Syllabus

- Course setup
  - Course description
  - Setup your PC
- ROS Essentials
  - ROS 2 - Basics
  - ROS 2 - Workspace
  - ROS 2 - Tools
  - ROS 2 - Simulation
  - ROS 2 - Sensors & controllers
- Manipulation
  - Kinematic and dynamic control
- Robot vision
  - Vision sensors
  - Computer vision
- Autonomous navigation
  - Autonomous navigation and path planning

Course setup

Course description

Aim of the course: introduce students to a variety of tools and concepts useful to robotic engineers. Use the Robot Operating System (ROS 2) to complete a robotics project

Prerequisites: there are no formal prerequisites. Basic knowledge of the following tools can make the life a bit easier

- Linux operating system (Ubuntu)
- Software versioning (Git)
- Programming languages (C/C++)
- Robot Operating System (ROS 2)
- Docker containers

Course description

Student learning outcomes

- Understand the ROS 2 architecture and tools
- Create ROS 2 C++ programs using libraries
- Develop applications for a robotic system

Job opportunities for robotics engineers

- Check out this video

Course description

Partecipation

- Hand-on classes in which *you learn by doing* are scheduled during the course. You have to be physically present to hand them in. Homeworks will be assigned on those days

Collaboration policy

- Collaboration on assignments is encouraged to share ideas and solve problems, but you must write your own code
- Students are expected to abide by the Honor Code. Honest and ethical behavior is expected at all times
- The work may be carried out individually or in a group of maximum 3-4 people. In the latter case, the work carried out by each group member must be clearly evidenced

Course description

Evaluation

- Four homeworks will be released throughout the course, they must be sent in the form of a report one week after release date. Each homework will count for 25%
- The final project also must be sent in the form of a report and discussed on the day of the exam
- Video of some previous years projects (2020, 2021, 2022, 2023)

Grading

- Homeworks: 50%
- Project: 25%
- Oral: 25%

Setup your PC

This course requires you to set up your PC. You'll find instructions on how to install and get familiar with the following computer software tools in a separate document (link). Here the main tools you must get familiar with

- **Linux**
  To develop ROS applications, you need to preferably work in a Ubuntu environment. Install Ubuntu 22.04 LTS from `https://ubuntu.com/`

- **ROS 2**
  Set of libraries and tools used to building robotics applications. You need to install ROS 2 and follow the preliminary tutorials on `https://www.ros.org/`

Setup your PC

- **Version control software (VCS)**
  Git is a VCS used to share code and keep track of it. Create an account on `https://github.com/` to host your Git repositories

- **Docker**
  Tool for creating and managing containers to run your applications anywhere. Visit `https://www.docker.com/` for more information

- **Computer programming**
  It is *highly recommended* to refresh your C++ skills using any C++ tutorial, e.g. `https://www.learncpp.com/`

ROS Essentials

History of ROS (Robot Operating System)

- Originally started in 2007 at the Willow Garage and Stanford Artificial Intelligence Laboratory under GPL license
- The goal was to establish a standard way to program robots while offering off-the-shelf software components easily integrable in custom robotic applications
- Since 2013 managed by Open-Source Robotics Foundation, now Open Robotics[1]
- De facto standard for robot programming in many university, companies etc.
- The goal of the ROS 2 project is to adapt to recent changes, leveraging what is great about ROS 1 and improving what isn't

[1] https://www.openrobotics.org/

ROS main features

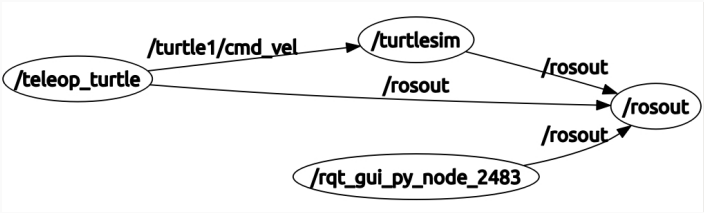
- Code sharing and reuse (do not reinvent the wheel)
- Distributed, modular design (nodes grouped in packages, scalable)
- Language independent (C++, Python, Java, ...)
- Individual programs communicate over defined API (ROS messages, services, etc.)
- Easy testing (ready-to-use)
- Vibrant community & collaborative environment
- Many robots are using ROS: `https://robots.ros.org/`

What is ROS?

- ROS is not an operating system rather a set of open source software libraries and tools that help you build robot applications
- From drivers to state-of-the-art algorithms, to user interfaces, ROS provides powerful developer tools that allow you to focus on the development of your robot application

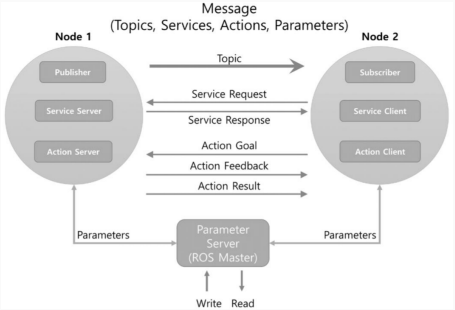
Plumbing (the computation graph)

- At its core, ROS provides a message-passing system, often called “middleware” or “plumbing”, that handles communication
- ROS processes are represented as nodes in a graph structure connected by edges by which they communicate via the ROS’s built-in and well-tested messaging system

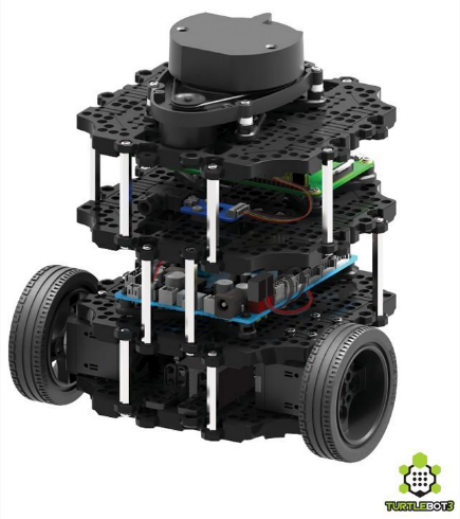


Plumbing (Cont'd)

- Nodes can publish or subscribe to named topics, can act as client or server for other nodes, or set or retrieve shared data from a communal database called the parameter server
- One node usually is a complex combination of publishers, subscribers, service servers, service clients, action servers, and action clients, all at the same time



Plumbing (Cont'd)

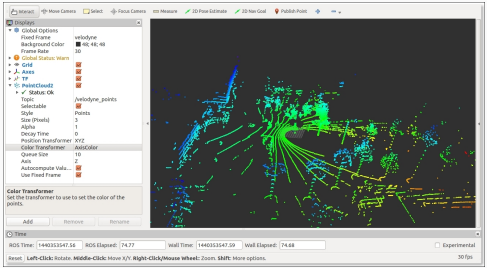


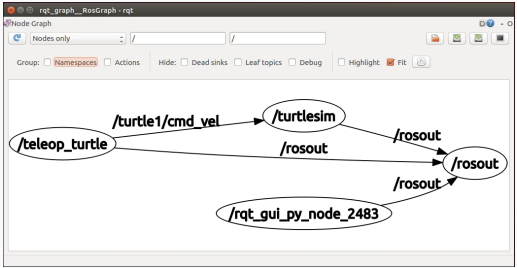
For instance, for the turtlebot

- a node retrieves laser data
- a node performs localization & mapping
- a node controls wheel motors
- a node gives velocity commands to the wheels
- ...

Tools

- Building robot applications is challenging. You have all the difficulties of any software development effort combined with the need to interact asynchronously with the physical world, through sensors and actuators
- ROS provides an extensive set of tools to configure, manage, debug, visualize, data log, and test your application





Capabilities

- ROS provides a broad collection of robot-agnostic libraries organized in packages that implement useful robot functionalities such as
  - the device driver for your GPS sensor
  - a walking and balance controller for your quadruped robot
  - a localization and mapping system for your mobile robot
- The goal of the ROS project is to continually raise the bar on what is taken for granted, and thus to lower the barrier to entry to building robot applications

Community

- ROS is supported and constantly improved by a large community of engineers and hobbyists from around the globe with a shared interest in robotics and open-source software
- Some useful links:
  - Tutorials - docs.ros.org
  - Demos - github.com/ros2/demos/
  - Examples - github.com/ros2/examples
  - Q & A site - answers.ros.org, robotics.stackexchange.com
  - Discussion - discourse.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-language: 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

CLI tools:

https://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools.html

Nodes

- Nodes are the processes that perform computation (executables)
- ROS nodes are written using ROS client libraries (available in different languages) implementing ROS functionalities such as communication between nodes
- Allow building multiple simple processes rather than a large process with all the functionality (modularity)
- A robot control system will usually comprise many nodes

ROS 1 vs ROS 2

- In ROS 1 there is a ROS Master, that takes care of connections and communication among nodes (TCP protocol)
- ROS 2 use the DDS, which mediates the peer-to-peer communication (decentralized) and guarantees more security and reliability
- ROS 1 nodes are single-process (*nodelets* are nodes running on the same process, useful when they share a lot of memory)
- ROS2 nodes can run on the same process and their lifecycle could be managed (state machine)
- ROS 2 client libraries (rclcpp and rclpy) share a common underlaying implementation (rcl). See here for more information.
- ROS 1 limits itself to Ubuntu or Debian. ROS 2 runs on macOS, Windows, real-time operating system, and other operating systems (microcontrollers)

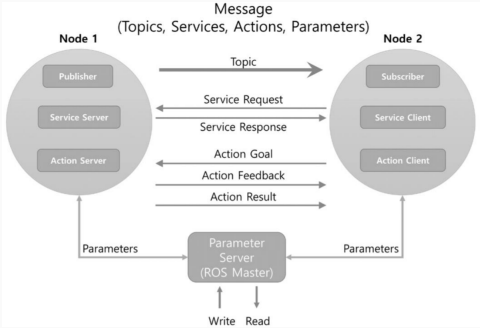
Discovery

- Discovery of nodes happens automatically through the underlying middleware of ROS 2 following the procedure:
  1. When a node is started, it advertises its presence to other nodes on the network with the same ROS domain (set with the `ROS_DOMAIN_ID` environment variable)
  2. Nodes periodically advertise their presence even after the initial discovery period
  3. Nodes advertise to other nodes when they go offline
- Nodes will only establish connections with other nodes if they have compatible Quality of Service settings

Inter-nodes communication

ROS nodes represent independent processes in the ROS stack, and they can communicate with each other using 3 primary modes:

- ROS Topics (publisher/subscriber)
- ROS Services (request/response)
- ROS Actions (action/feedback/result)



Example: `https://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Nodes/Understanding-ROS2-Nodes.html`

Interfaces

- Interfaces are a generic name for Topics, Services, and Actions
  - Topics have an associated Message Type that determines the layout of the message published to the topic
  - Services have an associated Service Type that determines the layout of the associated request and response
  - Actions have an associated Action Type that determines the layout of the request, result, and feedback
- ROS Interface Types are specified using the Interface Definition Language (IDL)

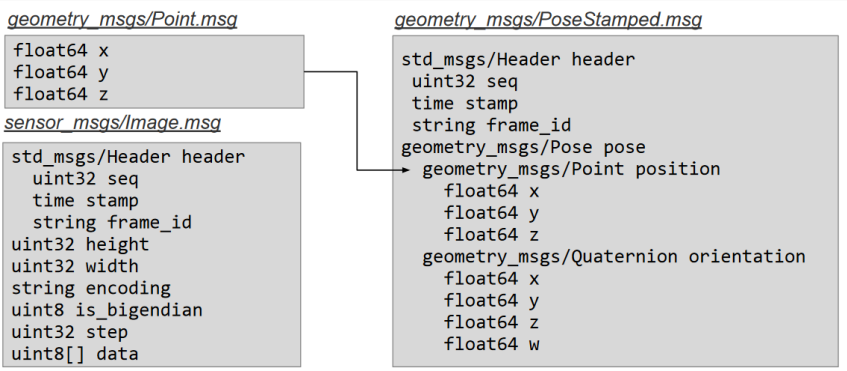
ROS 2 - Interfaces

Topics are labeled channels for communication between nodes and are an implementation of a Publish/Subscribe communication pattern

- A node can provide information by publishing a message to a topic
- A node can receive information by subscribing to a topic
- When a node publishes a message to a topic all nodes that have subscribed to that topic receive the message
- Topics are a many-to-many communication channel: any number of nodes may publish or subscribe to a given topic
- The message definition consists in a typical data structure composed by two main types: fields and constants
- Defined in *.msg files in the msg/ directory of a ROS package `<pkg>/msg/<MessageType>.msg`

ROS 2 message - example

- `geometry_msgs::PoseStamped` is used to share the timed pose of an object



Example: `https://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Topics/Understanding-ROS2-Topics.html`

Services

- Realize a Request/Response mechanism for inter-node communication
- A node provides a service by creating a service server and/or calls a service by creating a service client
- A service client sends a request to a service server and the service server replies by sending a response to the service client (one server - multiple clients)
- A service description file consists of a request and a response msg type, separated by - - - . Any two .msg files concatenated with a - - - are a legal service description
- Similar in structure to messages, services are defined in *.srv files in the srv/ directory of a ROS package `<pkg>/srv/<ServiceType>.srv` files

Example: `https://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Services/Understanding-ROS2-Services.html`

Actions

- Actions are a long-running (many seconds or minutes) task and receive periodic feedback and can be interrupted
- The action server receives a request from an action client (much like a service) and periodically sends feedback (over a topic) until the action is complete, whereupon it sends a result (like a service response)
- Like services, the request fields are before and the response fields are after the first triple-dash (- - -), respectively. There is also a third set of fields after the second triple-dash, which is the fields to be sent when sending feedback
- Action Types are stored in `<pkg>/action/<ActionType>.action` files

Example: `https://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Actions/Understanding-ROS2-Actions.html`

Parameters

- ROS parameters allows you to store and manipulate data on the ROS Parameter Server (that can be accessed by all ROS nodes)
- The Parameter Server can store integers, floats, boolean, dates, times, lists
- Oftentimes, parameters are set in Launchfiles, to provide each node you are starting with the proper configuration information
- Typically, nodes read parameters when they start, however, in ROS 2, a callback can respond to parameter changes
- Parameters are specified using the YAML format, YAML files can be stored on disk and loaded by rosparam CLI or a launchfile into the parameter server

Example: `https://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Parameters/Understanding-ROS2-Parameters.html`

ROS 2 - Launch

Launchfiles enable multiple nodes to be started with a single command

- In ROS 2 there are 3 formats for a launch file
  - Python: python scripts that use the ROS 2 launch API to configure and run nodes. The most flexible and powerful but also most complicated
  - XML: The format as in ROS 1. Directly declares what nodes are running but can perform minimal logic
  - YAML: Another format for writing what is essentially the same as an XML launchfile (do you like tags or indentation?)
- `ros2 launch` lets you run and interact with launchfiles
- Strive to have one launchfile completely start your project

Example: `https://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools/Launching-Multiple-Nodes/Launching-Multiple-Nodes.html`

ROS 2 - Recording data

Bags enable you to capture data from ROS topics to a file and play them back in real time

- Use `ros2 bag` to interact with and record bags
- Running robotics experiments is often frustrating and difficult. Capturing the data from a run and testing different algorithms and parameters on it is extremely useful
- `rqt_bag` is a plugin that enables interaction with bagfiles

ROS 2 - Recording data

Example: https://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools/Recording-And-Playing-Back-Data/Recording-And-Playing-Back-Data.html

ROS 2 - Logging

`rqt_console` is a GUI tool used to introspect log messages in ROS 2 in real time

- Nodes can log information at different logger levels, indicating the severity of the message
- There are five verbosity levels: DEBUG, INFO, WARN, ERROR, FATAL
- When running a node passing `--ros-args --log-level LEVEL` sets the logger level

Example: `https://docs.ros.org/en/iron/Tutorials/Beginner-CLI-Tools/Using-Rqt-Console/Using-Rqt-Console.html`

ROS 2 - Workspace

A **ROS 2 Workspace** is a directory containing a collection of ROS 2 packages

- Commonly, it contains the following folders:
  - `src`: the source code to ROS packages
  - `build`: a directory where intermediate files are stored
  - `log` files generated from building the packages
  - `install`: a directory where the packages are installed
- It's necessary to `source install/setup.bash` in your ROS 2 workspace. This makes ROS 2's packages available for you to use in that terminal
- You also have the option of sourcing an "overlay" - a secondary workspace where you can add new packages without interfering with the existing ROS 2 workspace that you're extending

The ROS 2 Workspace - building

- Workspaces must be built before they can be used

- Run `colcon build --symlink-install` from the main workspace directory to build all the packages in the `src` directory

Colcon

- `colcon` is the ROS 2 build-tool, which is used to build the workspace
- It is written in python and implemented as a series of extensions, which anyone can make to customize the build process
- As a build tool, colcon is capable of building projects that use many build systems: ament_python, ament_cmake, CMake (for C++), Catkin (for ROS1), setuptools (for python)
- colcon manages dependencies between multiple ROS packages written in different computer languages with different build systems

Colcon

- By default colcon builds packages in parallel
  - The dependencies specified in `package.xml` are used by colcon to build packages in the right order
  - If `<package A>` has a `build_depend` on `<package B>` then `colcon` always builds `<package B>` before `<package A>`
  - Your code may still compile by pure luck even if dependencies are specified indirectly. If `<package B>` finishes before the `<package A>` process needs it the build will succeed

colcon_cd

- `colcon_cd` allows you to quickly switch between the workspace directory and that of a package
- Install with `sudo apt install python3-colcon-cd`
- Then, add source `/usr/share/colcon_cd/function/colcon_cd.sh`
- From your workspace directory run `colcon_cd package` to go to that package
- You can then run `colcon_cd` to return to the workspace directory

colcon_clean

- `colcon_clean` allows you to easily remove the build results from a workspace
- Install with `sudo apt install python3-colcon-clean`
- Then, add source `/usr/share/colcon_cd/function/colcon_cd.sh`
- `colcon clean workspace` will clean all generated files
- `colcon clean packages` allows you to select individual packages to clean

ROS environment

- ROS relies on environment variables to control settings and find nodes and libraries
- ROS environment variables are set when underlay is sourced
  - To source the underlay run `source /opt/ros/<DISTRO>/setup.bash`
  - When you installed ROS you added the above command to the `.bashrc` so that it runs automatically whenever bash is opened
  - The underlay must be sourced to have access to the ROS command-line tools and system-installed packages

ROS environment (cont'd)

- Other ROS workspaces can be added by sourcing an `overlay`, which provides access to the packages installed in that overlay
  - When you `source install/setup.bash` after building the workspace you are adding the `overlay`, providing access to the packages you just built
  - Technically, you should not have the overlay sourced when using colcon build, which means you need a separate window for building and running ROS commands
  - Multiple ROS workspaces can be overlayed on top of each other allowing you to use packages from multiple workspaces or even override specific packages

Creating a workspace

https://docs.ros.org/en/humble/Tutorials/Beginner-Client-Libraries/Creating-A-Workspace/Creating-A-Workspace.html

ROS 2 - Packages

A **ROS package** is the organizational unit for your ROS 2 source code. It contains launch files, configuration files, message definitions, data, and documentation

- Binary versions of ROS packages are distributed on ROS's package server and can be downloaded via `apt`
- The naming convention when using apt is `ros-humble-<package-name>`
- With packages, you can release your ROS 2 work and allow others to build and use it easily (using Git for example)
- Package creation in ROS 2 uses `ament` as its build system and `colcon` as its build tool
- To create ROS code you need to create a package

Package structure

- ROS 2 Python and CMake packages each have their own minimum required contents:
  - `CMakeLists.txt` file that describes how to build the code within the package
  - `include/<package_name>` directory containing the public headers for the package
  - `package.xml` file containing meta information about the package
  - `src` directory containing the source code for the package
- A single workspace can contain as many packages as you want of different build types (CMake, Python, etc.)
- Best practice is to create your packages in the `src` folder within your workspace

Example: `https://docs.ros.org/en/iron/Tutorials/Beginner-Client-Libraries/Creating-Your-First-ROS2-Package.html`

The package.xml file

- All ROS packages have a base directory containing a manifest file called `package.xml`
  - This file is an XML document
  - The full specification for `package.xml` is in ROS Rep 149
  - The XML Schema for `package.xml` provides a machine-readable method for automatically validating the `package.xml`
- An important element of `package.xml` is the `<export><build_type>`, which determines the type of package
  - `ament python` is used for pure python packages
  - `ament_cmake` is used for C++ packages and packages that define custom Messages, Services, or Actions (i.e., Interfaces)
  - `ament_cmake_python` is for packages with mixed python/C++ code

The package.xml file

Required elements

- `<name>` The package name
- `<version>` The version number
- `<description>` A description of the package
- `<maintainer>` Authors
- `<license>` Ways the package may be distributed

Dependencies

- `<exec_depend>` Packages needed at runtime
- `<build_depend>` Packages needed at build time
- `<depend>` = `<exec_depend>` + `<build_depend>`

Ament CMake Packages

- `ament_cmake` is the build system for CMake based packages in ROS 2
- Ament Cmake packages are primarily used for C++ ROS projects
- Packages can be created using
  `ros2 pkg create --build-type ament_cmake <package_name>`

Custom Interfaces - definition

- Custom interfaces are created in their own `ament_cmake` type packages
- To create a custom interface file, first write an interface file using the ROS IDL
- The structure of an interface package looks like this

```
pkg_name/
├── package.xml           # The manifest file
├── CMakeLists.txt        # Build instructions, uses ament_* functions
├── msg/                  # Directory for custom message types
│   └── MessageType.msg   # A ROS IDL file defining a message
├── srv/                  # Directory for custom service types
│   └── ServiceType.srv   # A ROS IDL file defining a service
└── action                # Directory for custom action types
    └── ActionType.action # A ROS IDL file defining an action
```

Custom Interfaces - definition (cont'd)

- Edit `package.xml`
  - Add a `<buildtool_depend>` on `rosidl_default_generators`
  - Add an `<exec_depend>` on `rosidl_default_runtime`
  - You should also `<exec_depend>` and `<build_depend>` on any packages that use types defined by your custom interface

Custom Interfaces - usage

- To import in your package use
  `#include "<interface_package_name>/<msg|srv|action>/TypeName.hpp"`

- In python use
  `from <interface_package_name>.<msg|srv|action> import TypeName`

ROS 2 - Launch

Launch files allow you to start up and configure a number of executables containing ROS 2 nodes simultaneously

- ROS programs consist of many nodes communicating over topics and services, manually running them becomes tedious and hard to reproduce
- In ROS 2 (unlike ROS 1) there are multiple types of launchfiles:
  - Python launchfiles: python scripts that use the ROS 2 Launch API to declare what actions should be taken
  - XML (or YAML) launchfiles: they simply declare the nodes that should be running

XML Launch files

Example talker_listener_launch.xml

```
<launch>
  <node pkg="demo_nodes_cpp" exec="talker" output="screen" />
  <node pkg="demo_nodes_cpp" exec="listener" output="screen" />
</launch>
```

- `launch`: root element
- `node`: specifies ad node to be launched
- `name`: name of the node (free to be chosen)
- `pkg`: package containing the node
- `exec`: node type (there must be an executable with the same name)
- `output`: specifies where to output log messages (screen, log)

Launch arguments & parameters

- Create re-usable launch files with <arg> tag, which works like a parameter, e.g.
  `<arg name="arg_name" default="default_value"/>`
- Use arguments in launch file
  `$(arg arg_name)`
- When launching, arguments can be set
  `$ ros2 launch launch_file.xml arg_name:=value`
- The <param> tag allows for setting ROS parameters of a ROS node

Example

```
<node pkg="ros_demos" exec="publisher"> <param name="publish_frequency"
value="10"/> <remap from="generic_topic_name" to="my_topic"/> </node>
```

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

  `$(find-pkg-share package_name)`
- Pass arguments to the included file

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

Example

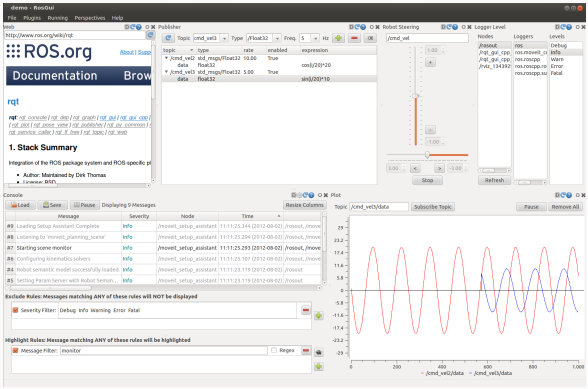
```
<include file="/opt/my_launch_file.py"/>
<include file="$(find-pkg-share my_pkg)/launch/some_launch_file.xml"/>
```

Work in a workspace

- Customize the simple publisher/subscriber
- Create a package
- Creating custom interfaces
- Using parameters in a class
- Customize launch file

rqt

- Qt-based GUI framework for ROS
- Various GUI tools in the form of plugins
- One can run all the existing GUI tools as dockable windows within `rqt`
- Users can create their own plugins for `rqt`



rqt - plugins

rqt_image_view: plugin for displaying images using `image_transport`

rqt_plot: plugin visualizing numeric values in a 2D plot using different plotting backends

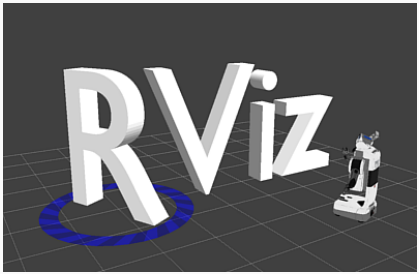
rqt_graph: plugin for visualizing the ROS computation graph

rqt_console: plugin for displaying and filtering ROS messages

rqt_logger_level: plugin for configuring the logger level of ROS nodes

Rviz2

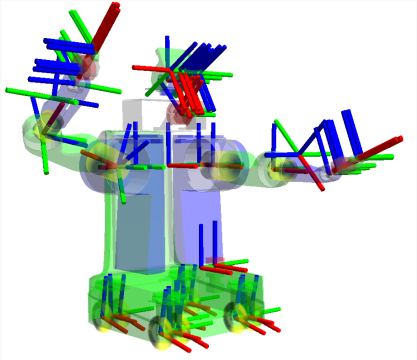
- 3D visualization tool for ROS
- Subscribes to topics and visualizes the message contents
- Different camera views (orthographic, top-down, etc.)
- Interactive tools to publish user information
- Save and load setup as RViz configuration
- Extensible with plugins



Rviz2 User guide: `https://docs.ros.org/en/humble/Tutorials/Intermediate/RViz/RViz-User-Guide/RViz-User-Guide.html`

Tf2

- `tf2` is a package that lets the user keep track of multiple coordinate frames over time
- `tf2` maintains the relationship between coordinate frames in a tree structure buffered in time
- `tf2` lets the user transform points, vectors, etc. between any two coordinate frames at any desired point in time



Example: `https://docs.ros.org/en/humble/Tutorials/Intermediate/Tf2/Introduction-To-Tf2.html`

Tf2 for your robot

- The `robot_state_publisher` package allows you to publish the state of a robot
- The package takes the joint angles of the robot as input and publishes the 3D poses of the robot links, using a kinematic tree model of the robot
- It uses the `URDF` specified by the parameter `robot_description` and the joint positions from the topic `joint_states` to calculate the forward kinematics of the robot and publish the results via `tf`
- Implemented as publisher/subscriber model on the topics `/tf` and `/tf_static`

Tf2 - Transform

- TF use a `tf2_ros::Buffer` to listen to all broadcasted transforms via `tf2_ros::TransformBroadcaster::sendTransform`
- Query for specific transforms between two coordinate frames in the transform tree via `lookupTransform()`
- The transform message is structured as follows

```
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
```

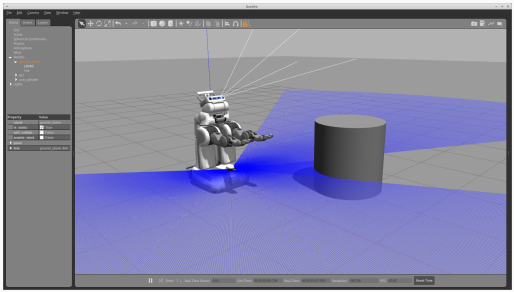
Tf2 - Command line

- Print info current transform tree
  `$ ros2 run tf2_ros tf2_monitor <frame1> <frame2>`
- Print info transform between two frames
  `$ ros2 run tf2_ros tf2_echo <frame1> <frame2>`
- Create a visual graph (PDF) `$ ros2 run tf2_tools view_frames`
- Run `rviz2` with `tf` enabled and begin viewing frames to see transforms

Example: `https://docs.ros.org/en/humble/Tutorials/Intermediate/Tf2/Writing-A-Tf2-Static-Broadcaster-Cpp.html`

Gazebo

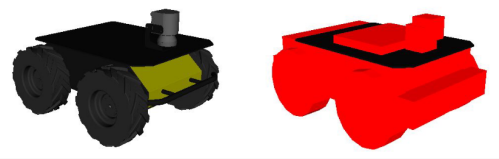
- Simulates 3D rigid-body dynamics
- Generates sensors' data including noise
- Realistic 3D visualization and user interaction
- Includes many robot models
- Provides a ROS interface
- Extensible with plugins
- Extensive command line tools



https://gazebosim.org/

Unified Robot Description Format - URDF

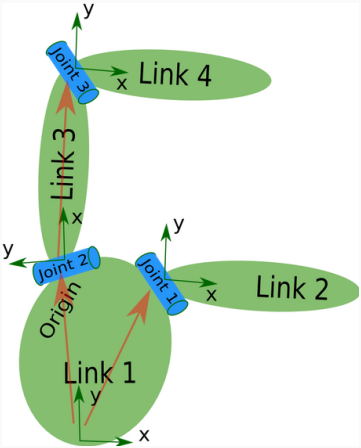
- Defines an XML format for representing a robot model
  - Kinematic and dynamic description
  - Visual representation
  - Collision model
- URDF generation can be scripted with XACRO macro



Visual meshes Primitives for collision

Unified Robot Description Format - URDF (cont'd)

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



The <robot> element

```
<robot name="robot_name">
  <!-- robot links and joints and more -->
  <link> ... </link>
  <link> ... </link>

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

The <joint> element

```
<joint name="my_joint" type="floating">
   <origin xyz="0 0 1" rpy="0 0 3.1416"/>
   <parent link="link1"/>
   <child link="link2"/>

   <calibration rising="0.0"/>
   <dynamics damping="0.0" friction="0.0"/>
   <limit effort="30" velocity="1.0" lower="-2.2" upper="
     0.7" />
   <safety_controller k_velocity="10" k_position="15"
     soft_lower_limit="-2.0" soft_upper_limit="0.5" />
</joint>
```

The <link> element

```
<link name="my_link">
  <inertial>
    <origin xyz="0 0 0.5" rpy="0 0 0"/>
    <mass value="1"/>
    <inertia ixx="100"  ixy="0"  ixz="0" iyy="100" iyz="0"
      izz="100" />
  </inertial>

  <visual>
    <origin xyz="0 0 0" rpy="0 0 0" />
    <geometry>
      <box size="1 1 1" />
    </geometry>
    <material name="Cyan">
      <color rgba="0 1.0 1.0 1.0"/>
    </material>
  </visual>

  <collision>
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <geometry>
      <cylinder radius="1" length="0.5"/>
    </geometry>
  </collision>
</link>
```

Unified Robot Description Format - URDF (cont'd)

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

```
robot_description = {"robot_description":
  robot_description_content}

joint_state_publisher_node = Node(
    package="joint_state_publisher_gui",
    executable="joint_state_publisher_gui",
)
robot_state_publisher_node = Node(
    package="robot_state_publisher",
    executable="robot_state_publisher",
    output="both",
    parameters=[robot_description],
)
rviz_node = Node(
    package="rviz2",
    executable="rviz2",
    name="rviz2",
    output="log",
    arguments=["-d", rviz_config_file],
)
```

Xacro

- Xacro is an XML macro language
- Include its namespace `xmlns:xacro="http://www.ros.org/wiki/xacro"` within the robot tag
- Used to construct shorter and more readable XML files by using macros
- It is heavily used in packages such as the urdf

Example Xacro

```
<xacro:macro name="cylinder_inertial" params="radius length
       mass *origin">
    <inertial>
      <mass value="${mass}" />
      <xacro:insert_block name="origin" />
      <inertia ixx="${0.0833333 * mass * (3 * radius *
      radius + length * length)}" ixy="0.0" ixz="0.0"
        iyy="${0.0833333 * mass * (3 * radius * radius +
      length * length)}" iyz="0.0"
        izz="${0.5 * mass * radius * radius}" />
    </inertial>
  </xacro:macro>

  <xacro:cylinder_inertial radius="${base_inertia_radius}"
      length="${base_inertia_length}" mass="${base_mass}">
       <origin xyz="0 0 0" rpy="0 0 0" />
  </xacro:cylinder_inertial>
```

Xacro (cont'd)

- Properties are named values or named blocks that can be inserted anywhere into the XML document
- Properties can be manually declared or loaded from YAML files

- Macros may contain other macros
- You can include other xacro files using the `xacro:include` tag

Xacro properties from yaml

```
<xacro:arg name="initial_pos" default="$(find
      arm_description)/config/initial_pos.yaml"/>

<xacro:property name="config_joint_limit_parameters" value=
      "${xacro.load_yaml(initial_pos)}"/>
```

Xacro include

```
<xacro:include filename="$(find package)/other_file.xacro"
      />
<xacro:include filename="other_file.xacro" />
<xacro:include filename="$(cwd)/other_file.xacro" />
```

Xacro (cont'd)

- Convert xacro to urdf from command line

```
$ ros2 run xacro robot_name.xacro -o robot_name.urdf
```

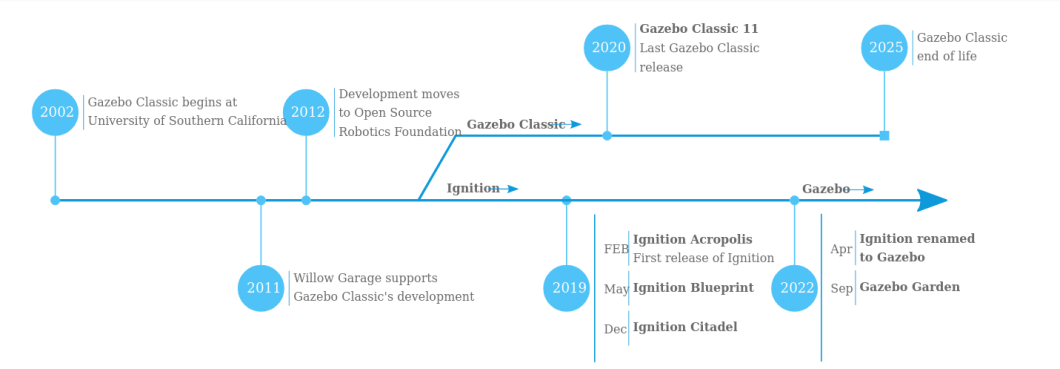
- ... or inside a launch file

```
Command(['xacro ', os.path.join(os.path.join(
get_package_share_directory('your_package')), "urdf", "robot.urdf.
xacro")])
```

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 automatically converts a URDF to SDF

Gazebo


2002
Gazebo Classic begins at University of Southern California
2012
Development moves to Open Source Robotics Foundation
2020
Gazebo Classic 11
Last Gazebo Classic release
2025
Gazebo Classic end of life
Gazebo Classic
Ignition
Gazebo
2011
Willow Garage supports Gazebo Classic's development
2019
FEB Ignition Acropolis
First release of Ignition
May Ignition Blueprint
Dec Ignition Citadel
2022
Apr Ignition renamed to Gazebo
Sep Gazebo Garden

Gazebo-simulators

- Gazebo can be launched by command line if you use fortress: `ign gazebo`
- Or if you use Harmonic: `gz sim`
- The vanilla command will launch the simulator as an empty world.
- Entities can be added to the scene by defining them using an XML-based file format called SDF (Simulation Description Format).
  `ign gazebo /path/to/sdf_file`
- Gazebo will parse the SDF file, reading it and converting the XML description of the entity into actual objects, models, and structures in the simulated world.

Sdf

- Syntax of sdf is very similar the one used for Urdf.
- Within the `<world>` tag we specify the models and plugin for the simulator
- Models are the entities composing the scene. Robots described in URDF are examples of models for a SDF file.

Example Sdf

```
<?xml version="1.0" ?>
<sdf version="1.6">
 <world name="box_world">
  <model name="box_model">
    <link name="box_link">
      <visual name="box_visual">
        <geometry>
          <box>
            <size>1 1 1</size>
          </box>
        </geometry>
      </visual>
    </link>
  </model>
  </world>
</sdf>
```

Rename the example as box.sdf, save it into your home directory and launch it by:

`ign gazebo box.sdf`

Gazebo-Ros integration

- ROS prescribes a specific way to launch all the pieces needed in your system. There are dedicated launch file and packages which support the integration of gazebo within the ros framework

launch Gazebo

```
gazebo_ignition_simulator = IncludeLaunchDescription(
    PythonLaunchDescriptionSource(
        [PathJoinSubstitution([FindPackageShare('ros_gz_sim'),
                               'launch',
                               'gz_sim.launch.py'])]), #ign_gazebo.launch.py
    launch_arguments={'gz_args': LaunchConfiguration('gz_args')}.items()
)
```

Spawn Urdf

```
gz_spawn_entity = Node(
        package='ros_gz_sim',
        executable='create',
        output='screen',
        arguments=['-topic', 'robot_description',
                   '-name', 'robot_name',
                   '-allow_renaming', 'true',
    )
```

Gazebo-Ros integration

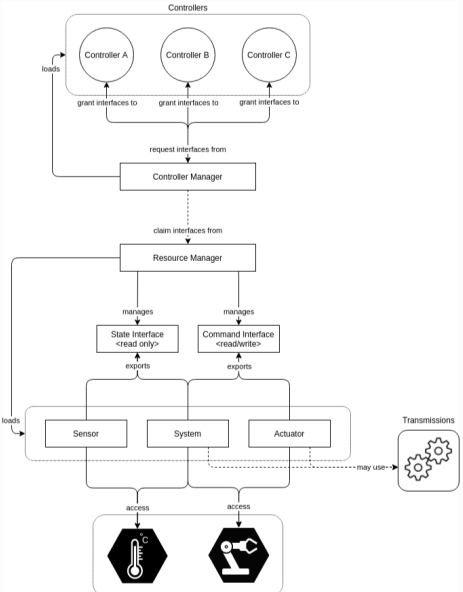
- Once the model has been spawned with the `create` node, it exists in the Gazebo environment but meshes could not be visualizable.
- Gazebo must be informed about the location of meshes through the environment variable `GZ_SIM_RESOURCE_PATH`.

Include these lines within the package.xml

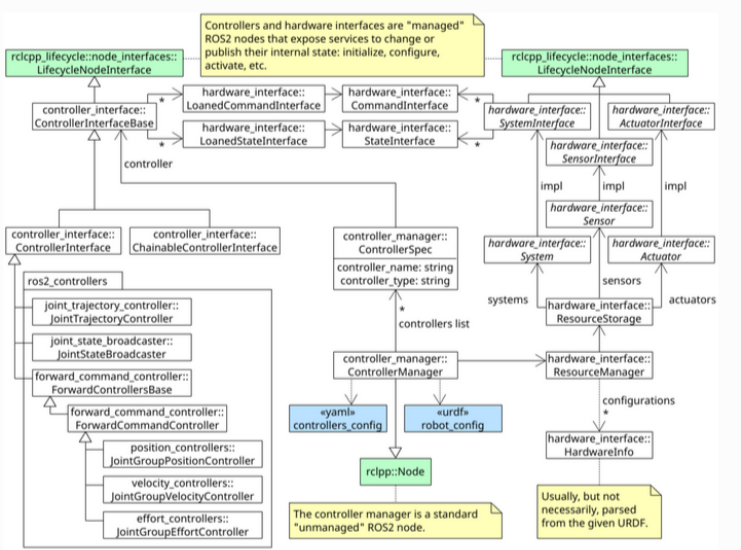
```
<export>
  <build_type>ament_cmake</build_type>
  <gazebo_ros gazebo_model_path="${prefix}/.." />
</export>
```

- The values in the attributes `gazebo_model_path` are appended to `GZ_SIM_RESOURCE_PATH`
- `${prefix}` refers to the path in the install directory, which also contains the meshes if specified in the CMake file.

ROS 2 - Sensors & controllers



ROS 2 - Sensors & controllers



Available controllers

- `joint_state_broadcaster` defined to publish joint states
- `joint_position_controller` position commands are used to control joint positions
- `joint_velocity_controller` velocity commands are used to control joint positions or velocities
- `joint_effort_controller` efforts commands are used to control joint positions, velocities or efforts
- `joint_trajectory_controllers` used to control the execution of joint-space trajectories on a group of joints

All the available controllers can be found in this repository: `https://github.com/ros-controls/ros2_controllers`

Configuring and launching controllers

Controllers are usually defined with YAML files. These files contain a list of controllers that the controller manager will load

Controllers are defined by a name and type

- The name is the identifier needed by the controller manager associated to the controller.
- The type represent the library that will be loaded

controllers.yaml

```yaml
controller_manager:
  ros__parameters:
    update_rate: 100 # Hz

    joint_state_broadcaster:
      type: joint_state_broadcaster/JointStateBroadcaster

    joint_trajectory_controller:
      type: joint_trajectory_controller/JointTrajectoryController

    position_controller:
      type: position_controllers/JointGroupPositionController
```

Configuring and launching controllers

ROS2 controllers whose type is specified within the YAML file, requires additional configurations

- Firstly, we need to specify the names of the joints on which those controllers will act
- Additional parameters can be configured depending on the type of controller

position controller.yaml

```yaml
joint_trajectory_controller:
  ros__parameters:
    joints:
      - joint_0
      - joint_1
      - joint_2
      - joint_3

    command_interfaces:
      - position

    state_publish_rate: 100.0
    action_monitor_rate: 20.0 # Defaults to 20
    allow_partial_joints_goal: true #
    open_loop_control: true
    allow_integration_in_goal_trajectories: true
```

Configuring and launching controllers

The YAML configuration file will be loaded using either a launch file or the URDF file.

- Use the URDF to load controllers if you are working in simulation; use the launch file otherwise
- We can both launch the Controller manager and the configurations including `gz_ros2_control` plugin
- If you are working with Ignition, the plugin is named `ign_ros2_control` plugin

Configuring and launching controllers

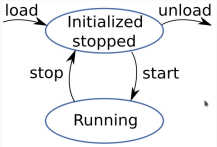
```
<gazebo>
  <plugin filename="ign_ros2_control-system" name="ign_ros2_control::IgnitionROS2ControlPlugin">
  <parameters>$(find arm_description)/config/pos_controller.yaml</parameters>
  <controller_manager_prefix_node_name>controller_manager</controller_manager_prefix_node_name>
  </plugin>
</gazebo>
```

- `<gazebo>` tag specify that we are are working within the Gazebo framework
- `<parameters>`: take as input a YAML file with the configuration of the controller
- `<controller_manager_name>`: Set controller manager name (default: `controller_manager`

If you launched the `robot-state-publisher` passing the urdf with the `ign_ros2_control` plugin, you should see the `/controller_manager` node to appear.

Controller management

Once the `controller_manager` node is active, we can load the ros2-controllers defined in the YAML file



Command line

```
$ ros2 run controller_manager spawner controller_name
```

Launch file

```
load_controller = Node( package="controller_manager", executable="
spawner", arguments=["name_controller", "--controller-manager", "/
controller_manager"], )
```

Using rqt

```
$ sudo apt-get install ros-<distro>-rqt-controller-manager
$ ros2 run crqt_controller_manager rqt_controller_manager
```

Controller management: Some useful commands:

- `$ ros2 control list_controller_types`: will print all the type of the available controllers that we could add inside the configuration YAML file
- `$ ros2 control load_controller --set-state active name_controller` : Load a new controller (you can load only active controllers)
- `$ ros2 control set_controller_state name_controller {inactive, active}`: Change the state of a controller
- `ros2 control unload_controller name_controller`

Hint
Use the `launch.action.RegisterEventHandler()` method to start the controllers after the model is spawned in Gazebo

Hardware Components

The hardware components realize communication to physical hardware and represent its abstraction in the ros2_control framework. There are three types of hardware

- system
- sensor
- actuator

which represent the hardware component. The ros2_control framework uses the `<ros2_control>`-tag in the robot's URDF file to describe its components: Components can be described through the ros2-controller interfaces:

- State Interfaces: to retrieve the states from the joints, actuators, or sensors
- Joint Command Interfaces: to send command to the actuators (sensors type haven't the command interface)

Hardware components: type sensors and actuators

The command and state interface can include position, velocity, and/or effort.

- Actuators type are very similar to the system type, but are related to only one joint. This type still provides both command and state interface for the single joint
- A sensor component is related to a joint (e.g., encoder) or a link (e.g., force-torque sensor). This component type has only reading capabilities

Hardware components: type system

With "system" type hardware components, we can specify multi-DOF robotic hardware, such as industrial robots. Within the `<ros2_control>`-tag, the `<joint>`-tag groups the interfaces associated with the joints of physical robots and actuators. These can be command and state interfaces to set the goal values for hardware and read its current state.

```
<ros2_control name="Name_of_the_hardware" type="system">
  <hardware>
    <plugin>library_name/ClassName</plugin>
    <param name="example_param">value</param>
  </hardware>
  <joint name="name_of_the_component">
    <command_interface name="interface_name">
      <!-- All of them are optional. ‘data_type‘ and ‘
      size‘ are used for GPIOs. Size is length of an
      array. -->
      <param name="min">-1</param>
      <param name="max">1</param>
      <param name="initial_value">0.0</param>
      <param name="data_type"></param>
    </command_interface>
    <state_interface name="position"/>
  </joint>
</ros2_control>
```

Hardware plugin

- Regardless of the type of hardware, we need to add a specific plugin to enable the hardware interface. For simulator environment, this plugin can be the `gazebo-ros-control` (for Gazebo classic), `gz-ros2-control` (for Gazebo harmonic), `ign-ros2-control` (for Gazebo fortress)

```
<hardware>
  <plugin>ign_ros2_control/IgnitionSystem</plugin>
</hardware>
```

Gazebo plugins

- A plugin is a chunk of code that is compiled as a shared library and inserted into the simulation
- Gazebo relies on plugins for rendering, physics simulation, sensor data generation, and many of the capabilities. Plugins make us control many aspects of the simulation like world, models, etc.
- This gives users great control and makes sure only what's crucial for a given simulation is loaded

Sensor plugins

- Gazebo Sensors provides a set of sensors models that can be configured at run time to mimic specific real-world sensors
- The use of a sensor requires adding the appropriate library to the project. We'll try now to add a camera sensor

```
<gazebo>
<plugin
  filename="gz-sim-sensors-system"
  name="gz::sim::systems::Sensors">
  <render_engine>ogre2</render_engine>
</plugin>
</gazebo>
```

camera sensor

Essential tags:

- `<name>` is the name of the entity that will appear in gazebo, and is specified by the user
- `<type>` specifies which kind of sensor we are using
- The tag `<topic>` represents the name of the Gazebo topic on which data will be published

```
<sensor name="camera" type="camera">
    <camera>
      <horizontal_fov>1.047</
      horizontal_fov>
      <image>
        <width>320</width>
        <height>240</height>
      </image>
      <clip>
        <near>0.1</near>
        <far>100</far>
      </clip>
    </camera>
    <always_on>1</always_on>
    <update_rate>30</update_rate>
    <visualize>true</visualize>
    <topic>camera</topic>
  </sensor>
```

camera sensor

The sensor is usually added to one of the links of our model

- In sdf it's enough to place this code within the `<link>` tag
- In urdf sensor should be added in another location of the file within the tag `<gazebo reference = "link_name">`
- Once the sensor has been added, a new topic will appear among the Gazebo topics. You can check all the available topic with: `ign topic -l`
- Check if the `/camera` topic appears (that's the name we defined in the `<topic>` tag)
- You can print the content of the topic with `ign topic -e -t /camera`
- With `ign topic -e -t /camera | less` you can print the header information

Ros-Gz-Sim

- The information from the sensor exists only within the Gazebo world
- We should make these informations available to ros nodes as well
- `ros_ign_bridge` provides a network bridge which enables the exchange of messages between ROS 2 and Gazebo. Its support is limited to only certain message types
- We can initialize a bidirectional bridge so we can have ROS as the publisher and Gazebo as the subscriber or vice versa. The syntax is `/TOPIC@ROS_MSG@GZ_MSG`, where `TOPIC` is the Gazebo internal topic, `ROS_MSG` is the ROS message type for this topic, and `GZ_MSG` is the Gazebo message type
- The name of the new ROS topic will be the same of the Gazebo topic. It can be changed by passing the new name as argument to the `parameter_bridge` node

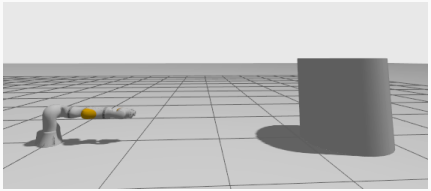
Ros-Ign-Sim

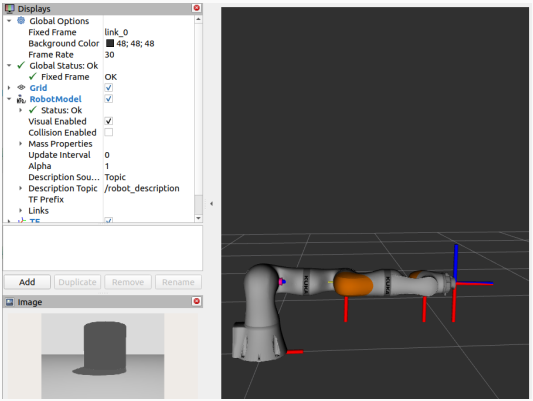
```
$ ros2 run ros_ign_bridge parameter_bridge /camera@sensor_msgs/msg/
Image@gz.msgs.Image
```

- The `ros2 run ros_ign_bridge parameter_bridge` command simply runs the `parameter_bridge` code from the `ros_ign_bridge` package
  - `/camera` is the name of the topic from which we want copy data
  - `sensor_msgs/msg/Image` is the message type that will be published on the Ros topic
  - `gz.msgs.Image` is the message type taken from the Gazebo topic
- The `@` indicates a bidirectional communications between the ROS and Gazebo.
- Once the command has been launched, the `/camera` topic will appear also among the ros2 topic list

Visualize Camera messages in Rviz

- Open Rviz with `rviz2`
- Add `by topic` (or by display) /camera/Image





Change the name of the ros topic

- The `parameter_bridge` node launched in that way will generate a new ros2 topic in which all the contents of the Gazebo topic declared before the first `@` symbol are published
- The new ros2 topic will have the same name of the Gazebo topic
- Here is shown how to launch the node by launch file and how to change the name of the ros2 topic

```
bridge_camera = Node(
    package='ros_ign_bridge',
    executable='parameter_bridge',
    arguments=[
        '/camera@sensor_msgs/msg/Image@gz.msgs.Image',
        '/camera_info@sensor_msgs/msg/CameraInfo@gz.
  msgs.CameraInfo',
        '--ros-args',
        '-r', '/camera:=/videocamera',
    ],
    output='screen'
)
```

Manipulation

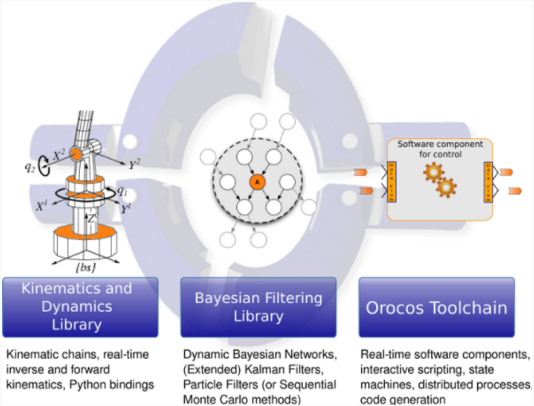
The OROCOS project

- Orocos (Open Robot Control Software) project aim was to create advanced C++ libraries for robot control
- Over the years, Orocos has become a large project of middleware and tooling for development of robotics software. The main parts of this project are
  - Orocos Real-Time Toolkit (RTT): a component framework that allows us to write real-time components in C++
  - Orocos Component Library (OCL): the necessary components to start an application and interact with it at run-time
- Orocos framework is well integrated with ROS, a popular software bundle with the largest community among roboticists to design new applications. Most of the concepts from both frameworks map well and are largely supported

The OROCOS project

- Additional libraries were also developed to complement the bundle for advance machine and robot control. These libraries include calculation of kinematic chains, filtering and advance task specification among others
  - Kinematics and Dynamics Library (KDL): an application independent framework for modeling and computation of kinematic chains
  - Bayesian Filtering Library (BFL): an application independent framework for inference in Dynamic Bayesian Networks, i.e., recursive information processing and estimation algorithms based on Bayes' rule
  - Reduced Finite State Machine (rFSM): a small and powerful statechart implementation in Lua.
  - Instantaneous Task Specification using Constraints (iTaSC): is a framework to generate robot motions by specifying constraints between (parts of) the robots and their environment.

The OROCOS project



The Kinematics and Dynamics Library (KDL)

- Orocos project to supply RealTime usable kinematics and dynamics code, it contains code for rigid body kinematics calculations and representations for kinematic structures and their inverse and forward kinematic solvers
  - Github: `https://github.com/orocos/orocos_kinematics_dynamics`
  - API: `http://docs.ros.org/en/indigo/api/orocos_kdl/html/index.html`
  - ROS: `https://wiki.ros.org/kdl`

The Kinematics and Dynamics Library (KDL)

- What can I use KDL for?
  - 3D frame and vector transformations: KDL includes excellent support to work with vectors, points, frame transformations, etc. You can calculate a vector product, transform a point into a different reference frame, or even change the reference point of a 6d twist
  - Kinematics and Dynamics of kinematic chains: You can represent a kinematic chain by a KDL Chain object, and use KDL solvers to compute anything from forward position kinematics, to inverse dynamics
  - Kinematics of kinematic trees: You can represent a kinematic chain by a KDL Chain object, and use KDL solvers to compute forward position kinematics. Currently no other solvers are provided

Robot vision

Introduction

Vision is a crucial aspect in robotics. It enables robots to perceive their environment extracting information from camera data

Applications include

- extracting an object and its position
- inspecting manufactured parts for production errors
- detecting pedestrians in autonomous driving applications
- make a robot arm perform a somewhat intelligent pick and place task

Interface the sensor

- To develop a program using camera sensors we need to interface them to the onboard computer of the robot
- This can be made mainly in two ways
  - Using operating system drivers
  - Vendor drivers
- Standard USB camera (like webcams) are directly accessible using low level routine provided by the operating system
- In Ubuntu/Linux, after plugging in the camera, check whether a `/dev/videoX` device file has been created using `$ ls /dev/ | grep video`

Interface the sensor

- If `ls /dev/ | grep video` does not list any file, you may have permission problems when trying to access the device
- We must be sure that our USER group is owner of the device. We can switch owner from `root` to `user` by for example `sudo chown root:user /dev/video0`
- If you are in a Docker container, you can use the `--privileged` option to the `docker run` command. This causes the device owner to be root
- To check if everything works and ROS can actually stream images, try `ros2 run usb_cam usb_cam_node_exe`. **Note**: The package `usb_cam` must be installed!

The `usb_cam` package

- `usb_cam` provides a configurable ROS Driver for standard V4L USB Cameras
- The source code is located at `https://github.com/ros-drivers/usb_cam`
- The package can be installed by
  ```
  $ sudo apt-get install ros-humble-usb-cam
  ```
- Launch the node provided with the package
  ```
  $ ros2 run usb_cam usb_cam_node_exe
  ```
- Show the image using `rqt_image_view` package
  ```
  $ ros2 run rqt_image_view rqt_image_view
  ```

The `usb_cam` package

- `usb_cam` publishes two important topics:
  - `/usb_cam/image_raw`: Uncompressed frames
  - `/usb_cam/camera_info`: Camera calibration matrices from the specified calibration YAML file provided by the camera's vendor, or obtained with tools from `camera_calibration` package
- Moreover several Services and Parameters are provided, check the documentation at `http://wiki.ros.org/usb_cam`
- The default configuration file is visible at: `https://github.com/ros-drivers/usb_cam/blob/develop/config/usb_cam.yml`
- Params can be changed ar runtime by means of the `rqt_reconfigure` by `$ ros2 run rqt_reconfigure rqt_reconfigure`

The `image_transport` package

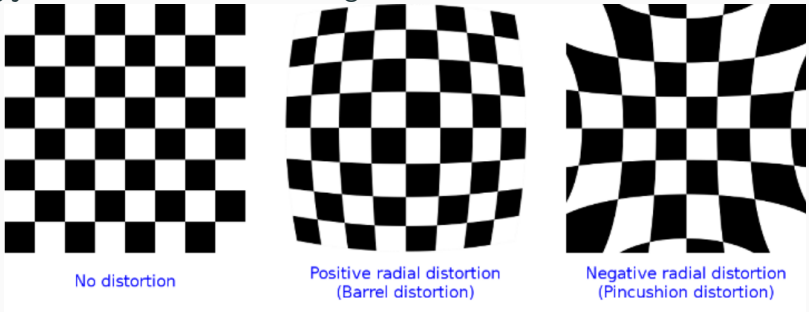
- The compressed format is useful to send images to other ROS nodes over the network or store video data into bagfiles
- These topics are published by the `image_transport` package that provides transparent support for transporting images in low-bandwidth compressed formats
- Its internal mechanism is very similar to using ROS Publishers and Subscribers, but specialized for images
- Check the documentation here: `http://wiki.ros.org/image_transport` and tutorials here: `https://github.com/ros-perception/image_transport_tutorials/tree/humble`

The `image_transport` package

- To use the compressed image we need to republish it in an uncompressed format, using the `republish` node of the `image_transport` package
- **Exercise:** Find the correct command line to execute this task

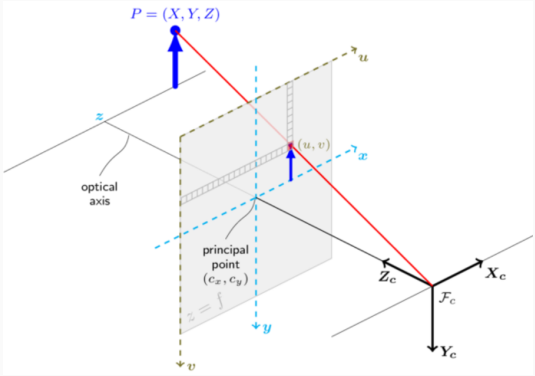
The calibration problem

- Cameras need to be calibrated to correct image distortions due to the camera's internal features
- We cannot make good use of an image produced by a fisheye-like lens without knowing just how it distorts the image



The calibration problem

Assuming a standard pinhole camera



$$s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

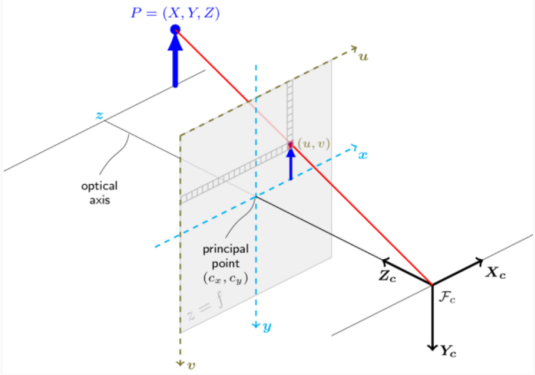
$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = R \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} + t$$

$$x' = x/z$$
$$y' = y/z$$
$$u = f_x * x' + c_x$$
$$v = f_y * y' + c_y$$

Simple, isn't it?

The calibration problem

Assuming a standard pinhole camera



$$s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = R \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} + t$$

$$x' = x/z$$

$$y' = y/z$$

$$x'' = x' \frac{1 + k_1 r^2 + k_2 r^4 + k_3 r^6}{1 + k_4 r^2 + k_5 r^4 + k_6 r^6} + 2p_1 x' y' + p_2(r^2 + 2x'^2)$$

$$y'' = y' \frac{1 + k_1 r^2 + k_2 r^4 + k_3 r^6}{1 + k_4 r^2 + k_5 r^4 + k_6 r^6} + p_1(r^2 + 2y'^2) + 2p_2 x' y'$$

where $r^2 = x'^2 + y'^2$

$$u = f_x * x'' + c_x$$

$$v = f_y * y'' + c_y$$

The calibration problem

- The `camera_calibration` package allows easy calibration of monocular or stereo cameras using a checkerboard calibration target[2]
- To start calibration in one console, use the following command:
  `$ ros2 run camera_calibration cameracalibrator --size 7x9 --square 0.015 --ros-args -r image:=/image_raw`
- The `size` option here denotes interior corners (e.g. a standard chessboard is 7×7), so for an 8×10 checkerboard, we go with 7×9

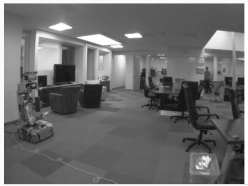
[2] You can create your own: `https://calib.io/pages/camera-calibration-pattern-generator`

The calibration problem

- You should now see the calibration window and begin the calibration process. Next, move the pattern to all screen corners and tilt in every direction. When enough information is gathered, press the calibrate button
- Now you can see your camera calibration data in the console. You can simply save it in a file with ini extension or press the save button in the app to save the same in a tarball with both ini and yaml format
- For more information follow instructions here: `https://docs.ros.org/en/rolling/p/camera_calibration/doc/tutorial_mono.html`

The calibration problem

- The `Camera Calibration Parser` helps you to create a yml file, which you can load with nearly all ros camera driver using the `camera_info_url` parameter
- The `image_proc` package removes camera distortion from the raw image stream
- It is meant to sit between the camera driver and vision processing nodes
- To perform rectification use: `$ ros2 run image_proc rectify_node`



Computer vision

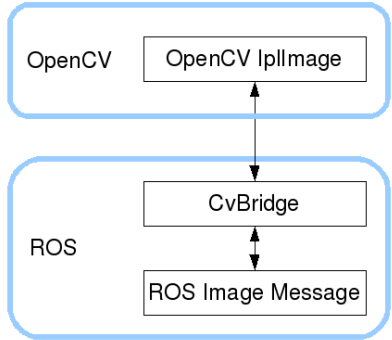
Computer vision is a field of computer science that focuses on enabling computers to identify and understand objects and people in images and videos

- ros2 `vision_opencv`[3] contains packages to interface ROS 2 with OpenCV which is a library designed for computational efficiency and strong focus for real time computer vision applications
- Interfacing ROS 2 with OpenCV is done via `cv_bridge` package
- To use OpenCV in your ROS code add this to your `CMakeLists.txt`:
  - `find_package(OpenCV)`
  - `include_directories (${OpenCV_INCLUDE_DIRS})`
  - `target_link_libraries(my_awesome_library ${OpenCV_LIBRARIES})`

[3] `https://github.com/ros-perception/vision_opencv`

OpenCV - CvBridge

- The CvBridge library converts between ROS 2 image messages and OpenCV image representation for perception applications
- ROS passes around images in its own `sensor_msgs/Image` message format
- Use CvBridge to convert ROS images into OpenCV `cv::Mat` format



OpenCV - CvBridge example code

- Create a new package with

```
$ ros2 pkg create ros2_opencv --dependencies rclcpp std_msgs
sensor_msgs cv_bridge image_transport OpenCV
```

- In your `CMakeLists.txt`:

```
add_executable(ros2_opencv_node src/ros2_opencv_node.cpp))
...
ament_target_dependencies(ros2_opencv_node rclcpp std_msgs
sensor_msgs cv_bridge image_transport OpenCV)
...
install(TARGETS ros2_opencv_node DESTINATION lib/$ PROJECT_NAME)
```

Fiducial Markers

- Efficient algorithms to perform object recognition and pose estimation working in real world environments are difficult to implement
- In many cases one camera is not enough to retrieve the three-dimensional pose of an object
- Markers are typically represented by a synthetic square image composed by a wide black border and an inner binary matrix which determines its unique identifier

Fiducial Markers

- When the *intrinsic parameters* of the camera and the size of the fiducial are known, the pose of the fiducial relative to the camera can be estimated
- The pose estimation code solves a set of linear equations to determine the world (X, Y, Z) coordinate of each of the vertices
- From this, we obtain the *transform* of the fiducial's coordinate system to the camera's coordinate system
- A robot can determine its position and orientation by looking at a number of fiducial markers –

Fiducial Markers

- To install the fiducial marker software packages ... search ROS 2 github code!

Fiducial Markers -The `aruco_ros` package

- To use the aruco_ros fiducial packages, clone it from the repo[4]
- Checkout the correct branch

```
$ cd ros2_ws/src/aruco_ros
$ git checkout humble-devel
```

- Compile and run to check if it works correctly

```
$ colcon build
$ ros2 launch aruco_ros single.launch.py
```

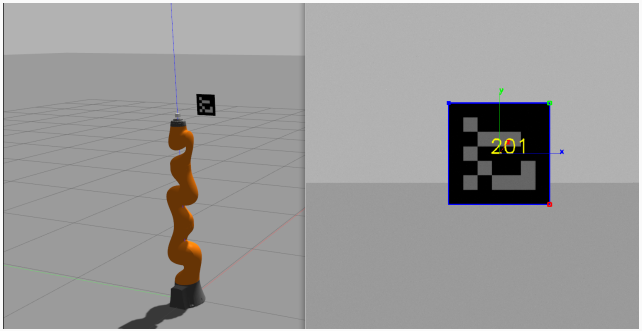
[4] https://github.com/pal-robotics/aruco_ros.git

Fiducial Markers - The `aruco_ros` package

- You should then subscribe to your camera topic
- **Exercise:** Create a launch file that starts the camera and connects the streamed `/image_raw` topic to the `aruco_ros_node`

Fiducial Markers

Let's test and use marker detectors in simulations using Gazebo ROS



Exercise: Create a launch file that starts the Gazebo simulator with a camera inside and the `aruco_ros_node`

Fiducial Markers

To use marker detectors in simulations using Gazebo ROS

- Generate the Aruco marker
  `https://chev.me/arucogen/`
- Create a Gazebo model, add it to the `GZ_SIM_RESOURCE_PATH`
  `$ export GZ_SIM_RESOURCE_PATH=<SOME_PATH>/gazebo_models/`

```
gazebo_models/
├ robot/
|  ├ meshes/
|  |  ├ mesh.stl
|  ├ model.config
|  ├ model.sdf
|
├ another_model/
|  ├ meshes/
|  |  ├ mesh.stl
|  ├ model.config
|  ├ model.sdf
...
```

Fiducial Markers

- Import the aruco model into your world, and save the world with name
- Relaunch the simulation loading the new world

```
<!-- path: <SOME_PATH>/gazebo_wor
<?xml version="1.0" ?>

<sdf version="1.6">
  <world name="robot_world">

    <include>
      <uri>model://robot</uri>
      <pose>0 0 0 0 0 0</pose>
    </include>

  </world>
</sdf>
```