# Robotics and ROS - Learn by Doing – Manipulators

*Udemy*

<https://www.udemy.com/course/robotics-and-ros-learn-by-doing-manipulators/learn/lecture/31597662?start=15#overview>

Ulisses’ Note: It still relies on ROS1, not ROS 2 ☹

See other equivalent book for ROS 2 !

# Introduction to the Course

## Course Presentation

## Meet your Teacher

## Boost your Robotics Software Developer Career

## Get the most out of the Course

## Course Material

## 6. Course Structure

Uses Arduinio UNO and Servo Motors SG 90



Get ROS to install in Ubuntu from ROS.org

## 7. Why an Operating System for the Robots

# Setup the Environment

## 8. Install Ubuntu on Virtual Machine

## 9. Install Ubuntu on Dual Boot

## 10. Install ROS

## 11. Install Modules and Dependencies

One useful module to install is “moveit”:

sudo apt install ros-jazzy-moveit

According to Claude:

*MoveIt is a powerful motion planning framework for robotic manipulation. Here are its main purposes:*

*Motion Planning*

* *Helps robots calculate collision-free paths to move their arms/manipulators from one position to another*
* *Handles complex environments with obstacles*
* *Plans smooth trajectories considering joint limits and kinematic constraints*

*Manipulation Tasks*

* *Pick and place operations*
* *Grasping objects*
* *Complex manipulation sequences*
* *Cartesian path planning (moving in straight lines in 3D space)*

*Robot Configuration*

* *Manages robot kinematics*
* *Handles different robot descriptions (URDF files)*
* *Manages collision checking between robot parts and environment*

*Key Features*

* *Integration with sensors for perception*
* *Real-time planning and replanning*
* *Support for multiple robots/arms*
* *Visual tools for setup and testing*
* *Inverse kinematics solving*
* *Path constraint handling*

*MoveIt is particularly popular in industrial robotics, research, and any application requiring precise robot arm control.*

Let’s continue with the package controllers:

sudo apt install ros-jazzy-ros2-control ros-jazzy-ros2-controllers

And then package gazebo:

sudo apt install ros-jazzy-ros2-control ros-jazzy-gz-ros2-control

sudo apt install ros-jazzy-ros-gz

Then package replacing *rosserial* (and *rosserial-arduino*) in ROS 2:

sudo apt install ros-jazzy-micro-ros-agent

Ulisses Note:

Info on how to install gazebo in ROS 2 Jazzy:

<https://gazebosim.org/docs/latest/ros2_gz_vendor_pkgs/>

To be able to use the gz command to run the usual commands, be sure that at least gz\_tools\_vendor package is installed. To have the gz command in the PATH, source the setup.bash from /opt/ros/${ROS\_DISTRO} as usual.

# Example running gz sim on Jazzy

export ROS\_DISTRO=jazzy

sudo apt-get install ros-${ROS\_DISTRO}-gz-tools-vendor ros-${ROS\_DISTRO}-gz-sim-vendor

. /opt/ros/jazzy/setup.bash

gz sim --help

And also: *roboticsgroup-upatras-gazebo-plugins*

Apparently we’ll need to get the src code for this one.

And *actionlib-tools:*

Actually, this is not needed in ROS 2, we can use the built-in “ros2 action” for the same purpose.

We also need python libraries:

1st lets install *pip*:

sudo apt install python3-pip python3-rosdep python3-colcon-common-extensions

It’s recommended to install module for virtual environments:

sudo apt install python3-venv

Create a virtual env:

python3 -m venv .venv # Creates a virtual environment named ".venv"

Activate the virtual environment:

source .venv/bin/activate

Then:

pip install flask

pip install flask-ask-sdk

pip install ask-sdk

Another useful tool is *ngrok*

In the site dashboard.ngrok.com. follow setup and installation section.

This will open our computer to be accessible by the internet, like if it was an web server.

Here it is used Visual Studio Code, with extensions for C++, Python and CMake tools.

## 12. <OPTIONAL>Using Docker</OPTIONAL>

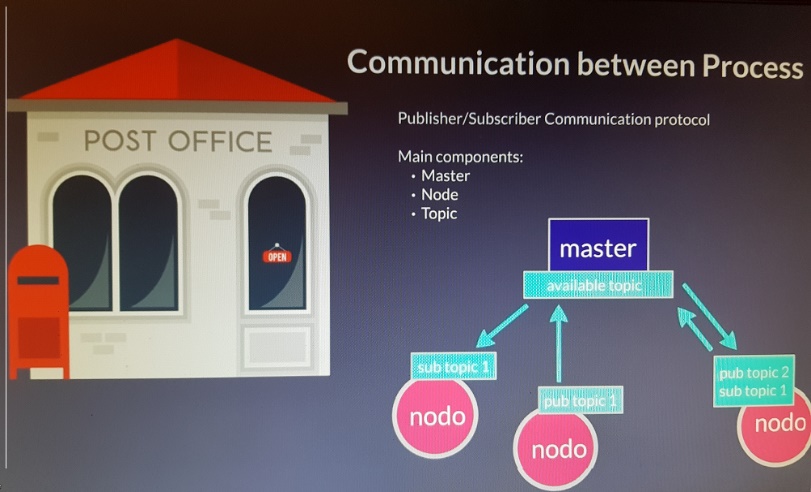
# Introduction to ROS

## 13. What is ROS

## 14. Hardware Abstraction

## 15. Low-Level Device Control

## 16. Messaging between Process



A node is simply an application that executes a functionality (it’s almost like a process).

Uses the protocol publish-subscriber.

The “master” is like the post office.

But the receivers must go to the post-office to get the message.

## 17. Package Management

A package can contain 1 or more nodes.

## 18. Architecture of a ROS Application

A ROS application is composed by:

* Workspace
* Package
* Node

The *Workspace* is the folder that contains all the source code.

A *package* could contain everything about controlling motors, for instance.

A *node* is a more elementary component.

## 19. <LAB>Create the first ROS Project</LAB>

Let’s create the workspace:

mkdir -p arduinoboot\_ws/src

Here we’ll create our ROS packages.

To make it a ROS workspace run:

catkin\_make

This is a build tool to generate the exes from src code

Then, to create a package:

catkin\_create\_pkg arduinobot\_test roscpp rospy std\_msgs

For ROS to be aware of our package(s) run:

source setup.bash

(inside the “devel” subfolder)

Then we will be able to run commands like

roscd arduinobot\_test

A python code should import this:

/usr/bin/env python3

import rospy

from std\_msgs.msg import String

if \_\_name\_\_ == ‘\_\_main\_\_’:

rospy.init\_node(“simple\_publisher\_py”, anonymous=True)

pub = rospy.Publisher(‘chatter’, String, queue\_size=10)

rate = rospy.Rate(10) // 10Hz frequency

counter = 0

while not rospy.is\_shutdown():

hello\_msg = “Hello World %d” % counter

pub.publish(hello\_msg)

rate.sleep()

counter += 1

We’ll also need to change the CMakeLists.txt file.

Before running our application we must start the *master*.

roscore

After “sourcing” our node

source devel/setup.bash

we can run it with

rosrun arduinobot\_test simple\_publisher.py

simple\_publisher.py is the node

We can get the list of currently active topics with:

rostopic list

To see which messages are passing through the “chatter” topic:

rostopic echo /chatter

## 20. <PY>Create a Publisher Node</PY>

We create the python code inside a new “scripts” folder inside “arduinoboot\_test”

## 21. <C++>Create a Publisher Node</C++>

## 22. <PY>Create a Subscriber Node</PY>

#!/usr/bin/env python3

import rospy

from std\_msgs.msg import String

def callback(data):

# This function is called each time a new message is published on the topic /chatter

# The message that has been published is then passed as input to this function

rospy.loginfo("I heard %s", data.data)

if \_\_name\_\_ == '\_\_main\_\_':

# Initialize a ROS node called listener

rospy.init\_node('listener', anonymous=True)

# register a subscriber on the topic /chatter that will listen for String messages

# when a new message is received, the callback function is triggered and starts its execution

rospy.Subscriber("chatter", String, callback)

# keeps the node up and running

rospy.spin()

the most important function here is *rospy*.***Subscriber*** . When a new message is received it will invoke the method “callback”

## 23. <C++>Create a Subscriber Node</C++>

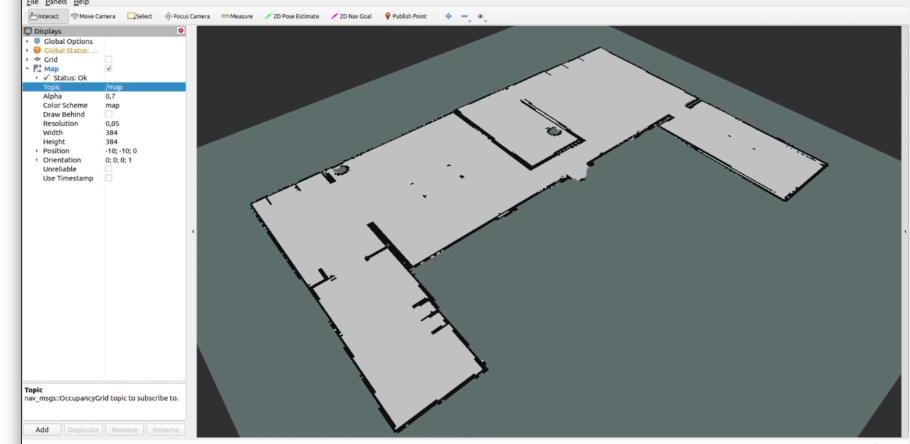
# Digital Twin

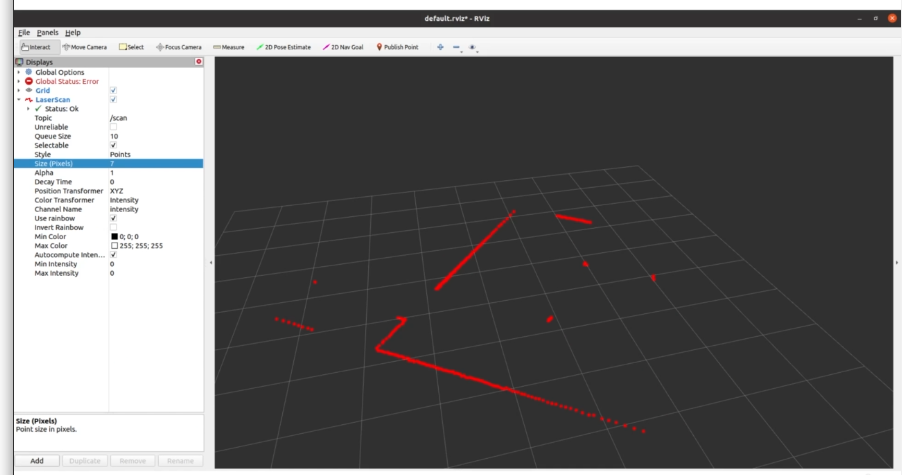
## 24. What is a Digital Twin

A digital twin is a tool to simulate (and debug) the robot in the PC.

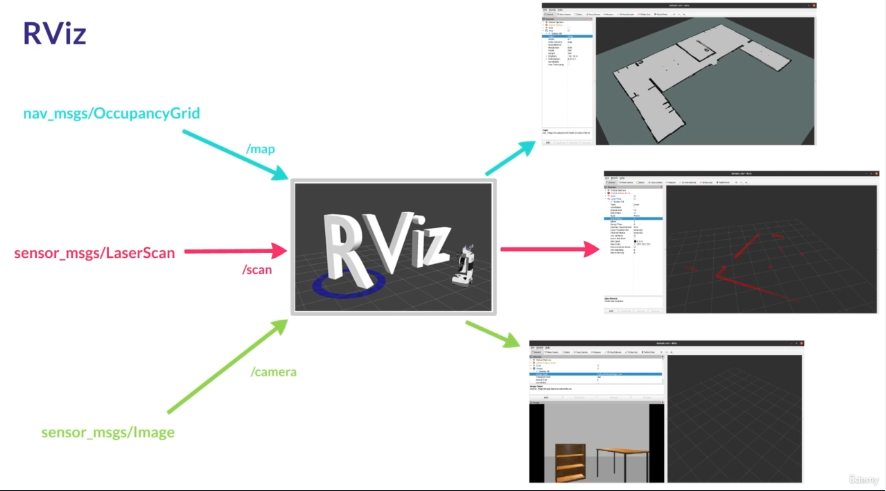
## 25. RViz

RViz stands for ROS Visualisation.

Above we visualise a map the robot has to navigate (selecting the ”/map” topic)#

We can also visualise readings from sensors. For example, we can also select “/scan” topic to visualize the laser distance scans.

We could also visualize images from cameras.



The message that contains a map is called ***occupancy grid***.

## 26. URDF

## 27. <LAB>Create the Digital Twin</LAB>

Here we’ll create an URDL xml file for ROS.

There’s a bit of Xacro macros as well.

## 28. <LAB>Complete the Digital Twin</LAB>

Continuation with URDFs, joints, links ...

<link name="base\_link">

<visual>

<origin rpy="0 0 0" xyz="-0.5 -0.5 0"/>

<geometry>

<mesh filename="package://arduinobot\_description/mesh/basement.STL"

scale="0.01 0.01 0.01"/>

</geometry>

</visual>

</link>

We can use the ***multiplier*** attribute of <mimic> to have the joint with the mimic to move in the opposite direction of the original one:

<joint name ="joint\_5" type="revolute">

<parent link="claw\_support"/>

<child link = "gripper\_left"/>

<origin xyz="-0.22 0.13 -0.1"/>

<axis xyz="0 0 1"/>

<limit lower="0" upper="${PI/2}" effort="${effort}" velocity="${velocity}"/>

<mimic joint="joint\_4" multiplier="-1" offset="0"/>

</joint>

### Mimic tag according to Claude:

Th**e <mimic>** tag in ROS URDF (Unified Robot Description Format) is used to create a dependent joint that mirrors or scales the movement of another joint. This is particularly useful when you want to create joints that move together in a coordinated fashion without having to control them separately.

Let me explain how the <mimic> tag works and its key attributes:

#### Basic Structure

When added to a joint definition in a URDF file, the <mimic> tag looks like this:

<joint name="follower\_joint" type="revolute">  
 <!-- joint properties -->  
 <mimic joint="master\_joint" multiplier="1.0" offset="0.0"/>  
</joint>

#### Key Attributes

The <mimic> tag has three important attributes:

1. **joint** (required): This specifies the name of the master joint that will be mimicked. The follower joint will move in relation to this master joint.
2. **multiplier** (optional, default=1.0): This is a scaling factor that determines how much the follower joint moves in relation to the master joint. For example:
   1. A multiplier of 1.0 means the follower joint moves exactly like the master joint
   2. A multiplier of 2.0 means the follower joint moves twice as much as the master joint
   3. A multiplier of -1.0 means the follower joint moves in the opposite direction to the master joint
3. **offset** (optional, default=0.0): This adds a constant offset to the position calculation. The final position is calculated as:

follower\_position = multiplier \* master\_position + offset

#### Practical Example

Here's a complete example of mimicking joints in a robot hand, where finger joints move together:

<robot name="hand\_example">  
 <!-- Base link -->  
 <link name="palm"/>  
   
 <!-- First finger links -->  
 <link name="finger1\_proximal"/>  
 <link name="finger1\_distal"/>  
   
 <!-- Master joint (proximal finger joint) -->  
 <joint name="finger1\_proximal\_joint" type="revolute">  
 <parent link="palm"/>  
 <child link="finger1\_proximal"/>  
 <axis xyz="0 0 1"/>  
 <limit lower="0" upper="1.57" effort="10" velocity="1.0"/>  
 </joint>  
   
 <!-- Follower joint (distal finger joint that mimics proximal) -->  
 <joint name="finger1\_distal\_joint" type="revolute">  
 <parent link="finger1\_proximal"/>  
 <child link="finger1\_distal"/>  
 <axis xyz="0 0 1"/>  
 <limit lower="0" upper="1.57" effort="10" velocity="1.0"/>  
 <mimic joint="finger1\_proximal\_joint" multiplier="0.8" offset="0.1"/>  
 </joint>  
</robot>

In this example, whenever the finger1\_proximal\_joint moves, the finger1\_distal\_joint will automatically move at 80% of the rate (multiplier=0.8) plus a slight offset (0.1 radians). This creates a natural finger-curling motion where the distal joint follows the proximal joint's movement with a slight modification.

## Implementation Details

When you use mimic joints in ROS, you need to be aware of a few important points:

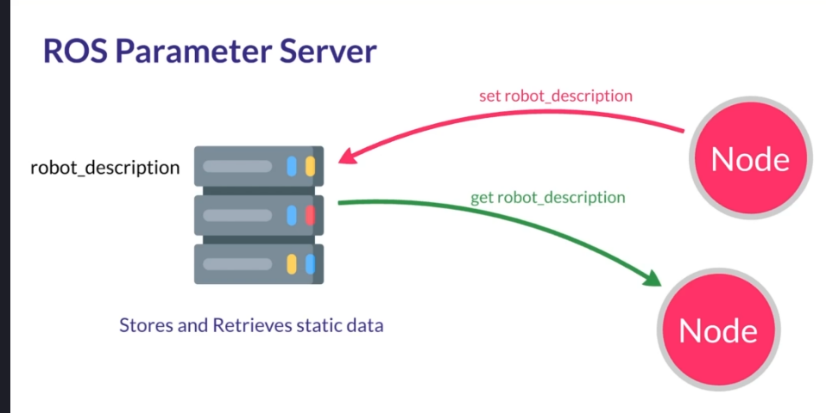
1. The gazebo simulator implements mimic joints automatically.
2. If you're using ros\_control, you need to load the joint\_state\_controller and the JointGroupPositionController or similar controllers.
3. In your YAML configuration file for controllers, you only need to specify the master joints as controlled joints. The mimic joints will follow automatically.
4. Behind the scenes, the mimicking is handled by the gazebo\_ros\_control plugin, which applies the appropriate calculations during simulation.

This approach greatly simplifies control of complex mechanisms like multi-joint grippers, parallel linkages, or differential drive systems, as you only need to command the master joints and the dependent joints will follow accordingly.

## 29. ROS Parameter Server

This offers additional functionality to the nodes. In this case to store and shares variables/values (also called *parameters*) between them.

In the case below we are **set**ting the variable named “robot\_description” using the command **set**. Other node can get its value using the ***get*** command.



## 30. <LAB>ROS Parameter Server</LAB>

In the video it says to run “roscore” to start the ros master, but we don’t need this in ros2

### What replaces roscore in ros2 according to Claude

In ROS 2, there is no direct equivalent to "roscore" because the architecture has fundamentally changed to eliminate this central point of coordination. This represents one of the most significant architectural differences between ROS 1 and ROS 2.

In ROS 1, roscore was a collection of nodes and programs that were prerequisites for a ROS-based system to run. It included:

* The ROS Master (providing name registration and lookup)
* The Parameter Server (storing system-wide parameters)
* rosout (for centralized logging)

ROS 2 replaces this centralized architecture with a distributed discovery system based on DDS (Data Distribution Service). This architectural shift offers several advantages:

The DDS middleware in ROS 2 handles discovery automatically through a protocol called Simple Discovery Protocol (SDP). When a node starts, it broadcasts its presence to the network, and other nodes discover it without needing a central coordinator. This happens through multicast communication rather than through a central server.

This distributed approach offers significant benefits:

* Improved robustness, as there's no single point of failure
* Better support for multi-robot systems
* Enhanced real-time performance capabilities
* Simplified network setup for distributed systems

In practical terms, when working with ROS 2, you simply launch your nodes, and they discover each other automatically. You don't need to start a special service first like you did with "roscore" in ROS 1.

For example, if you want to run a simple publisher-subscriber system in ROS 2:

1. You can start a publisher in one terminal:
2. ros2 run demo\_nodes\_cpp talker
3. And a subscriber in another terminal:
4. ros2 run demo\_nodes\_cpp listener

These nodes will find each other automatically without needing a roscore. Under the hood, DDS handles the discovery, addressing, and data exchange protocols that were previously managed by the ROS Master.

If you're transitioning from ROS 1 to ROS 2, this is one of the key mental shifts you'll need to make—there's no central service to start before running your nodes, which simplifies deployment but requires understanding the distributed nature of ROS 2's communication system.

### Parameters I ros2 - Who takes care of storing them and retrieving their values?

In ROS 2, parameter management has been completely redesigned while maintaining some conceptual similarities to ROS 1. Unlike ROS 1's centralized Parameter Server, ROS 2 uses a distributed parameter system where each node manages its own parameters locally.

Parameters in ROS 2 are owned and stored by individual nodes. When you set or get parameters, you're directly interacting with specific nodes rather than a centralized server. This aligns with ROS 2's broader philosophy of distributed architecture and elimination of single points of failure.

Here's how parameters work in ROS 2:

Each node maintains its own parameter server, storing parameter values in its own memory space. When you launch a node, it initializes with default parameters or ones provided at startup. These parameters exist only within that node's lifecycle—when the node terminates, those parameter values are no longer available unless explicitly saved.

For parameter access and modification, ROS 2 provides several methods:

1. Command line interface for human operators:

# List parameters for a node

ros2 param list /my\_node

# Get a specific parameter value

ros2 param get /my\_node my\_parameter

# Set a parameter value

ros2 param set /my\_node my\_parameter 42

1. API calls from other nodes using parameter services:

# Example in Python using the rclpy client library

import rclpy

from rcl\_interfaces.srv import GetParameters

# Create a client to get parameters from another node

get\_parameters\_client = node.create\_client(

GetParameters, '/other\_node/get\_parameters')

1. Parameter callbacks within the node itself:

# Setup a parameter callback in your node

def parameter\_callback(parameters):

for param in parameters:

node.get\_logger().info(f'Parameter changed: {param.name}')

return SetParametersResult(successful=True)

node.add\_on\_set\_parameters\_callback(parameter\_callback)

Despite this distributed approach, ROS 2 provides mechanisms for global parameter management:

1. **Parameter files**: You can load parameters for multiple nodes from YAML files at launch time.

# params.yaml

node\_1:

ros\_\_parameters:

param1: 10

param2: "hello"

node\_2:

ros\_\_parameters:

param3: 3.14

1. **Launch files**: These can set parameters for multiple nodes during system startup.

# Example launch file setting parameters

from launch import LaunchDescription

from launch\_ros.actions import Node

def generate\_launch\_description():

return LaunchDescription([

Node(

package='my\_package',

executable='my\_node',

name='custom\_node',

parameters=[{'my\_param': 42}]

)

])

1. **Parameter services**: Each node automatically creates services allowing other nodes to get, set, and list its parameters.

For centralized parameter viewing, tools like ros2 param list can enumerate parameters across all running nodes, giving the appearance of a unified parameter space, though behind the scenes these parameters remain distributed.

This distributed parameter approach offers advantages in resilience and scalability while presenting some challenges for global configuration management compared to ROS 1's centralized parameter server. The ecosystem continues to evolve with tools designed to simplify parameter management across complex distributed systems.

In the video it says to use “rosparam list”, but since we are using ros 2, we should run:

ros2 param list

## 31. <LAB>View the Robot in RViz</LAB>

In the video it says to run "rusrun xacro xacro arduinobot\_ws/src/arduinobot\_description/urdf/arduionobot.urd.xacro" but in ros2 it's different:

In ROS 2, the equivalent command for processing XACRO files has changed to use the xacro executable directly through the command line. The ROS 2 version of your command would be:

xacro arduinobot\_ws/src/arduinobot\_description/urdf/arduionobot.urd.xacro

If you want to save the processed URDF to a file (which is often what you want to do), you would use:

xacro arduinobot\_ws/src/arduinobot\_description/urdf/arduionobot.urd.xacro > arduinobot.urdf

### Using Command Line for Parameter Operations in ROS2 accordign to Claude

You can interact with parameters using ROS 2 command-line tools:

List all parameters for a node:

bashCopyros2 param list /my\_node

Get a specific parameter value:

bashCopyros2 param get /my\_node string\_param

Set a parameter value:

bashCopyros2 param set /my\_node int\_param 200

Save all parameters from a node to a file:

bashCopyros2 param dump /my\_node > my\_node\_params.yaml

Load parameters from a file to a running node:

bashCopyros2 param load /my\_node my\_node\_params.yaml

*(end of Claude explanation)*

To launch a node (in ros1 was “rosrun robot\_state\_publisher robot\_state\_publisher”) . In ros2 we should run:

ros2 run robot\_state\_publisher robot\_state\_publisher

### Explanation of command containing robot\_state\_publisher according to Gemini:

ros2 run: This is the ROS 2 command-line tool used to execute a specific node within a package. It replaces rosrun from ROS 1.

* **robot\_state\_publisher** (first instance): This is the name of the package containing the executable.
* **robot\_state\_publisher** (second instance): This is the name of the executable within the robot\_state\_publisher package.

**What robot\_state\_publisher does:**

The robot\_state\_publisher node plays a crucial role in a ROS-based robot system. Its primary function is to publish the current kinematic state of the robot based on the joint positions it receives. It takes information about the robot's joints (typically from a sensor or a simulation) and calculates the 3D pose of each link in the robot's kinematic chain.

**Parameters and Their Uses:**

The robot\_state\_publisher node has several parameters that can be configured. Here are some of the common ones and their purposes:

* **robot\_description** (String, usually loaded from a launch file):

Purpose: This is the most crucial parameter. It specifies the content of the robot's URDF file. This parameter is typically loaded from a param tag within a launch file, reading the URDF from a file or a string.

Example (in a launch file):

<param name="robot\_description" command="$(find xacro)/xacro --inorder '$(find my\_robot\_description)/urdf/my\_robot.urdf.xacro'"/>

This example uses the xacro tool to process a URDF.xacro file and sets the resulting XML string to the robot\_description parameter.

* **publish\_frequency** (Double, default: 50.0 Hz):

Purpose: This parameter controls how frequently the robot\_state\_publisher publishes the transform information on the /tf topic. A higher frequency means more frequent updates, which can be important for real-time applications but might consume more computational resources.

* **tf\_prefix (String, default: "")**:

Purpose: This parameter allows you to add a prefix to all the frame names published by the robot\_state\_publisher. This can be useful when you have multiple robots in the same environment to avoid naming collisions in the /tf tree. For example, if you set tf\_prefix to robot1\_, the frame named base\_link would be published as robot1\_base\_link.

* **use\_tf\_static** (Boolean, default: False):

Purpose: This parameter determines whether the fixed transforms (those that don't change over time, defined in the URDF) should be published on the /tf\_static topic instead of the regular /tf topic. In ROS 2, it's generally recommended to publish static transforms on /tf\_static for efficiency.

* **frame\_prefix** (String, default: ""):

Purpose: Similar to tf\_prefix, this parameter adds a prefix to the frame names. It might be used interchangeably with tf\_prefix depending on the specific ROS version or package usage.

robot\_state\_publisher is a predefined node that comes as part of a standard ROS package.

Other examples of predefined nodes include:

* **rviz**: For visualization.
* **joint\_state\_publisher**: For manually or programmatically publishing joint states (often used in simulations or simple setups).
* Various sensor drivers (e.g., for cameras, lidar).
* Navigation stack components (e.g., **map\_server**, **move\_base**)

Run the following command:

$ ros2 run joint\_state\_publisher\_gui joint\_state\_publisher\_gui

This will open the *Joint State Publisher* window

run the following command:

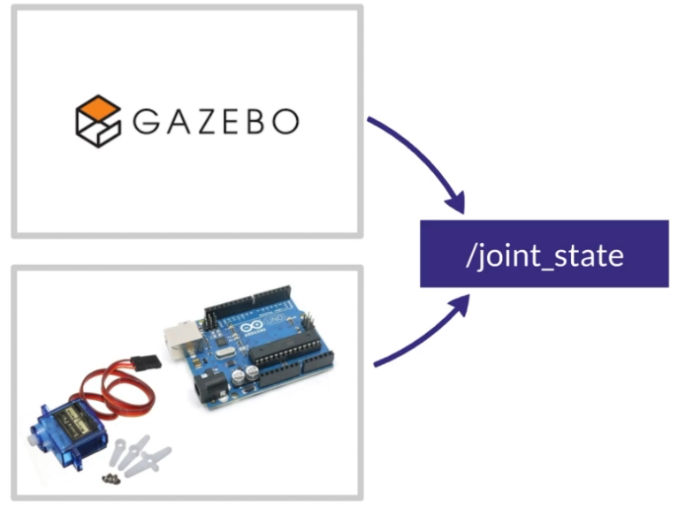
$ ros2 run rviz2 rviz2

This will open an empty RViz

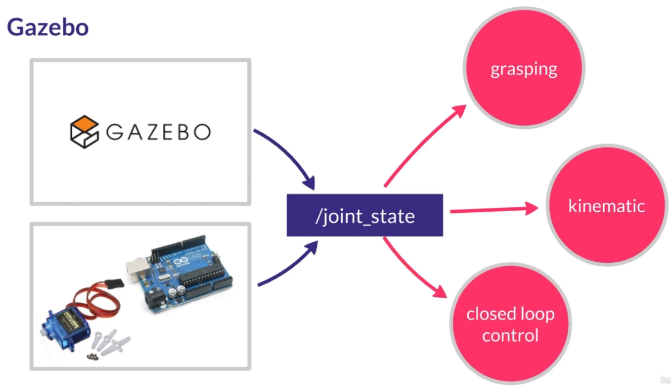
We can then use the “***File / Open Config***” menu option to load an existing configuration. Let’s load “display.rviz” config file.

## 32. Gazebo

The Gazebo simulator and the real hardware will publish info in the same topic.



This way, all the other applications that subscribe and use these messages won’t see any difference.



## 33. <LAB>URDF for the Simulation</LAB>

Let’s add physical properties like inertia that will be used in Gazebo.

## 34. ROS Launch File

## 35. <LAB>Launch the Visualization</LAB>

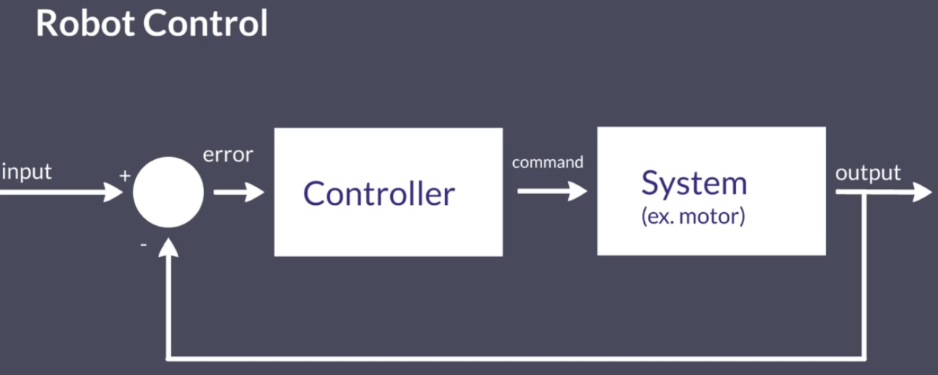
## 36. <LAB>Launch the Simulation</LAB>

# Control

## 37. ROS Control

Let’s start by looking at the ***actuation*** of the motors.

The action of sending a ***command*** to a motor and following its result is called ***control***.



The ***input*** is the target we want to get to.

The input is compared with the current status of the system.

The ***error*** variable is the difference between the *output* and the *input*, tells us how far is the current status is far from its desired value.

There is a library ros control for this.

***ros2\_control according to ChatGpt***

The ros2\_control library in ROS 2 is a framework for robot hardware abstraction, real-time control, and controller management. It provides a structured way to interact with hardware interfaces while enabling efficient communication between robot hardware and controllers.

To use ros\_control in ROS 2 Install the necessary packages:

sudo apt install ros-jazzy-ros2-control ros-jazzy-ros2-controllers

### ROS control library (according to Grok)

The ROS Control library in ROS 2 (Robot Operating System 2) is a framework designed to provide a standardized way to interface with robot hardware, manage controllers, and facilitate real-time control of robotic systems. It builds on the concepts introduced in ROS 1's `ros\_control` package but has been adapted and improved for ROS 2's architecture, leveraging its modern communication middleware (DDS) and enhanced tools.

Here’s an overview of the ROS Control library in ROS 2:

**Key Features**

1. **Hardware Abstraction**:

- ROS Control provides a `hardware\_interface` layer that abstracts the robot’s hardware (e.g., motors, sensors, actuators). This allows developers to write controllers that are hardware-agnostic, making code reusable across different robots.

- You define a `RobotHW` class specific to your hardware, which exposes joint states, commands, and other interfaces (e.g., position, velocity, effort).

2. **Controller Management**:

- The `controller\_manager` is a central component that loads, unloads, starts, and stops controllers dynamically at runtime. It ensures that controllers can access the hardware interface and operate in a coordinated manner.

- Controllers are typically implemented as plugins, making it easy to extend functionality.

3. **Real-Time Capabilities**:

- ROS Control is designed to work in real-time environments. It integrates with real-time schedulers (e.g., in Linux with PREEMPT\_RT) and ensures predictable control loops.

- The update loop runs at a fixed frequency, where sensor data is read, controller logic is executed, and commands are written to the hardware.

4. **Standard Controller Types**:

- ROS Control includes a set of pre-built controllers, such as:

- **Joint Position Controller**: Commands joint positions.

- **Joint Velocity Controller**: Commands joint velocities.

- **Joint Effort Controller**: Commands torques or forces.

- **Joint Trajectory Controller**: Executes smooth trajectories for multiple joints.

- These can be customized or extended for specific use cases.

5. **Integration with ROS 2**:

- Unlike ROS 1, where `ros\_control` relied on ROS topics and services, ROS 2’s version uses the ROS 2 ecosystem, including nodes, lifecycle management, and DDS-based communication.

- It integrates with tools like `ros2\_control`, which is the ROS 2-specific implementation of the control framework.

**Components**

- **ros2\_control**: The core framework in ROS 2, providing the infrastructure for hardware interfaces, controllers, and the controller manager.

- **ros2\_controllers**: A package containing standard controller implementations (e.g., trajectory controllers, PID controllers).

- **hardware\_interface**: Defines how the robot’s hardware communicates with the control layer.

- **controller\_manager**: Orchestrates the controllers and ensures they run in sync with the hardware.

**How It Works**

1. **Hardware Interface**: You implement a hardware interface for your robot, specifying how to read sensor data (e.g., joint positions) and write commands (e.g., motor torques).

2. **Controller Implementation**: You load controllers (either pre-built or custom) that process the sensor data and compute commands.

3. **Controller Manager**: This runs the control loop, calling each active controller’s `update` method at a specified rate.

4. **ROS 2 Integration**: The system communicates with other ROS 2 nodes (e.g., for trajectory planning or visualization) via topics, services, or actions.

**Example Use Case**

Imagine a robotic arm with 6 joints. You’d:

- Write a `**hardware\_interface**` to read joint encoder positions and send torque commands to the motors.

- Load a `**joint\_trajectory\_controller**` to follow a smooth path from a start to an end position.

- Use the `**controller\_manage**r` to start/stop the controller and switch to a `joint\_effort\_controller` if you need force control instead.

**Getting Started**

- **Installation**: In ROS 2 (e.g., Humble or Iron distributions), you can install `ros2\_control` and related packages via:

sudo apt install ros-<distro>-ros2-control ros-<distro>-ros2-controllers

- **Documentation**: The official ROS 2 documentation (e.g., on `docs.ros.org`) provides detailed tutorials, including how to set up a hardware interface and configure controllers.

- **Examples**: The `ros2\_control\_demos` repository on GitHub offers practical examples to explore.

**Differences from ROS 1**

- **Lifecycle Nodes**: Controllers in ROS 2 can use lifecycle management for better state handling (e.g., configure, activate, deactivate).

- **DDS**: Communication is more robust and scalable compared to ROS 1’s TCP-based system.

- **Modularity**: ROS 2’s `ros2\_control` is more modular and easier to integrate with other tools like MoveIt 2.

If you’re working on a specific project or have a particular aspect of `ros2\_control` you’d like to dive deeper into (e.g., writing a custom controller, troubleshooting real-time performance), feel free to let me know, and I can tailor the explanation further!

## 38. YAML Configuration File

## 39. <LAB>Load the Configuration file Parameter Server</LAB>

## 40. ROS Timer

## 41. <PY>Create a ROS Timer</PY>

## 42. <C++>Create a ROS Timer</C++>

## 43. ROS Services

## 44. <PY>Create a Service Server</PY>

## 45. <C++>Create a Service Server</C++>

## 46. <PY>Create a Service Client</PY>

## 47. <C++>Create a Service Client</C++>

## 48. Angles Converter Service

## 49. <PY>Angles Converter Service Server</PY>

## 50. <C++>Angles Converter Service Server</C++>

## 51. <LAB>Create the Controller</LAB>

## 52. <LAB>Complete the Controller</LAB>

## 53. <LAB>Configure the Controller</LAB>

## 54. <LAB>Launch the Controller</LAB>

# Kinematics

Robot Kinematics

2D Frames - Translation

2D Frames - Rotation

2D Frames - RotoTranslation

3D Frames - Translation

3D Frames - Rotation

3D Frames - RotoTranslation

Forward Kinematics

<LAB>The TF Package</LAB>

Inverse Kinematics

MoveIt!

<LAB>MoveIt! Setup</LAB>

<LAB>Configure MoveIt!</LAB>

# Applications

Robot Applications

ROS Actions

Actionlib Interface

<PY>Create an Action Server</PY>

<C++>Create an Action Server</C++>

<PY>Create an Action Client</PY>

<C++>Create an Action Client</C++>

<PY>Application Action Server</PY>

<C++>Application Action Server</C++>

# Alexa Integration

Alexa Skill

Develop Alexa Skills

<LAB>Test Alexa Skills</LAB>

<LAB>Interface the Robot with Alexa</LAB>

<LAB>Voice Interaction Model</LAB>

<LAB>Complete Simulation of the Robot</LAB>

# Buid the Robot

Mechanics and Electronics

Arduino and ROS

<HWLAB>Create a Publisher Node with Arduino</HWLAB>

<HWLAB>Create a Subscriber Node with Arduino</HWLAB>

<HWLAB>Servomotors with Arduino</HWLAB>

<HWLAB>Robot Control with Arduino</HWLAB>

<HWLAB>Launch the Complete Robot</HWLAB>

# Conclusions

Summary

What's next?

[BONUS]: Continue Learning