UNIVERSITÀ
DI TRENTO
Department of
Information Engineering and Computer Science

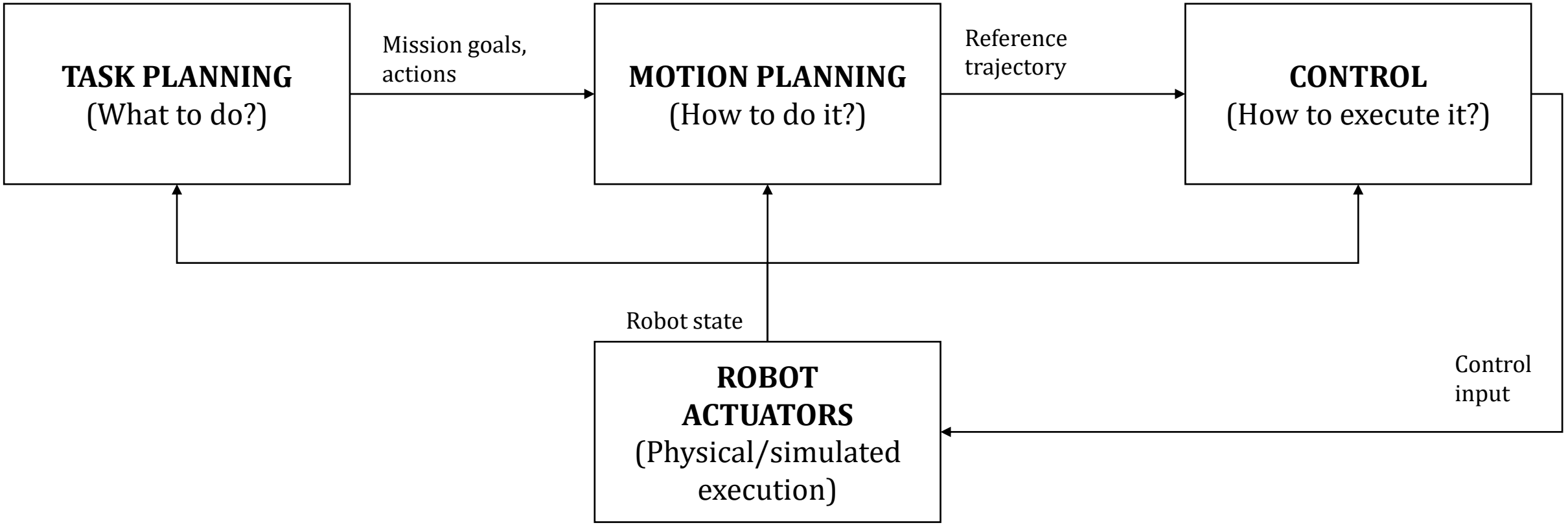
ROS 2 Control

Edoardo Lamon

Software Development for Collaborative Robots

Academic Year 2025/26

High-level Framework for a Robotic Application

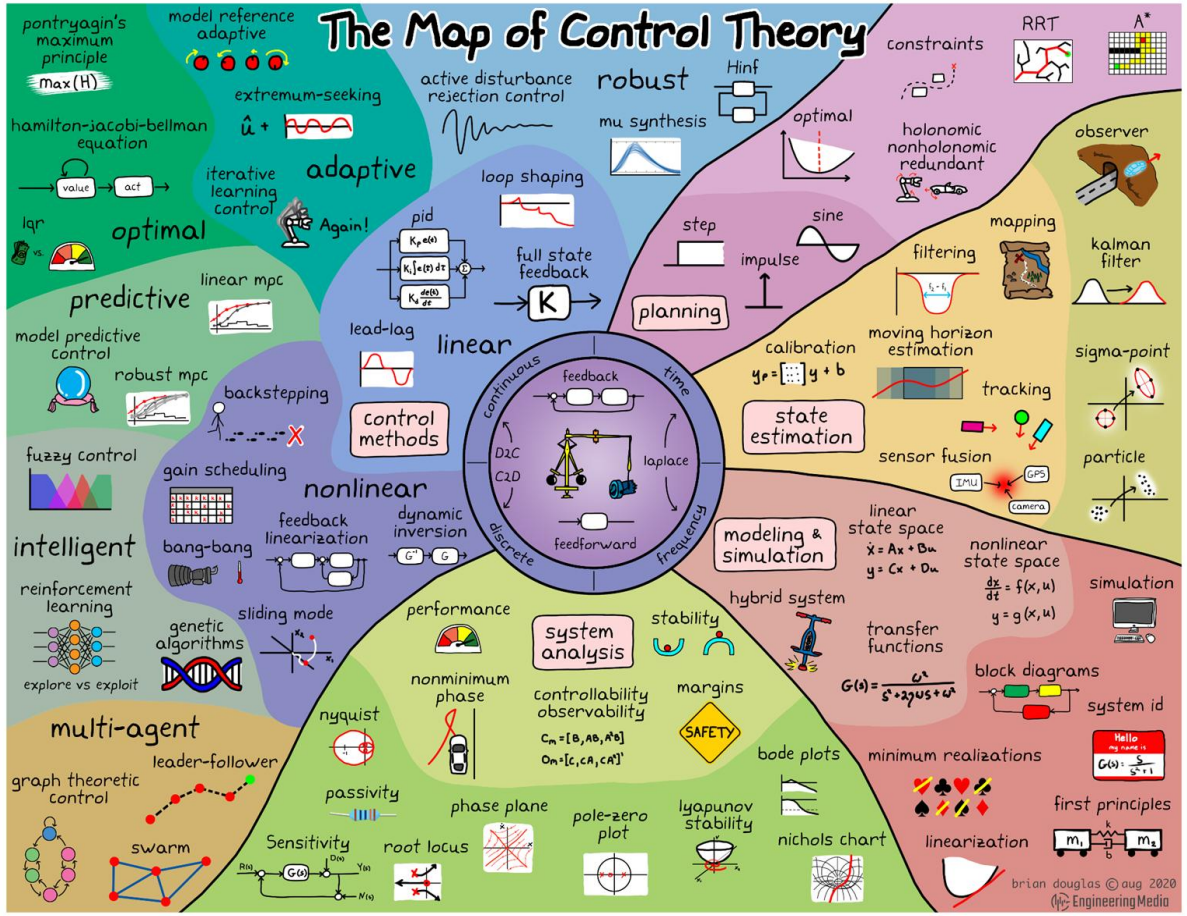


High-level Framework for a Robotic Application

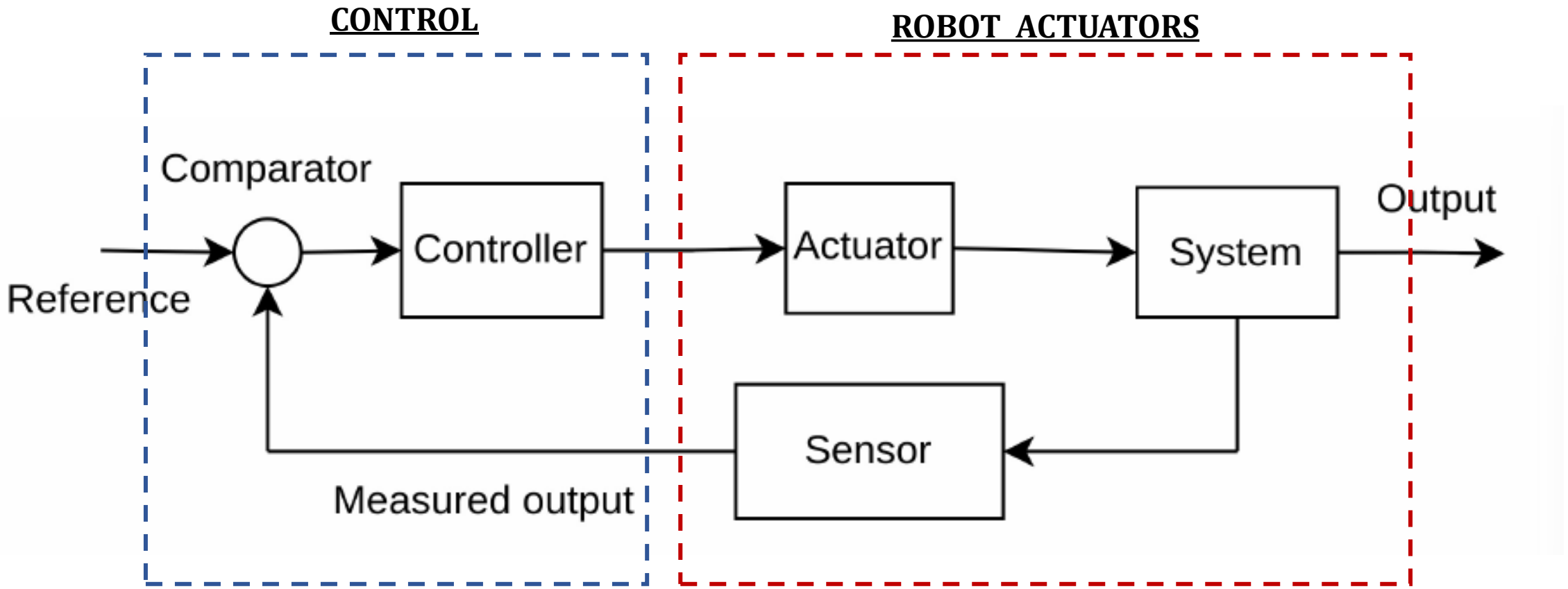
Layer	Purpose	Time Scale	Typical Input	Typical Output
Task Planning	Decides *what* to do: high-level goals and action sequences to achieve a mission.	Seconds → Minutes	Mission objective, world model, available actions	Sequence of actions or goals
Motion Planning	Decides *how* to move to reach each goal: finds a feasible path.	Milliseconds → Seconds	Start/goal states, environment map, robot kinematics	Cartesian of joint trajectory/path
Control	Executes *how to move:* converts planned motion into motor commands while reacting to sensors.	Milliseconds	Reference trajectory, sensor feedback	Low-level torque/ velocity/position commands

ROS 2 Tools

Controllers in ROS 2



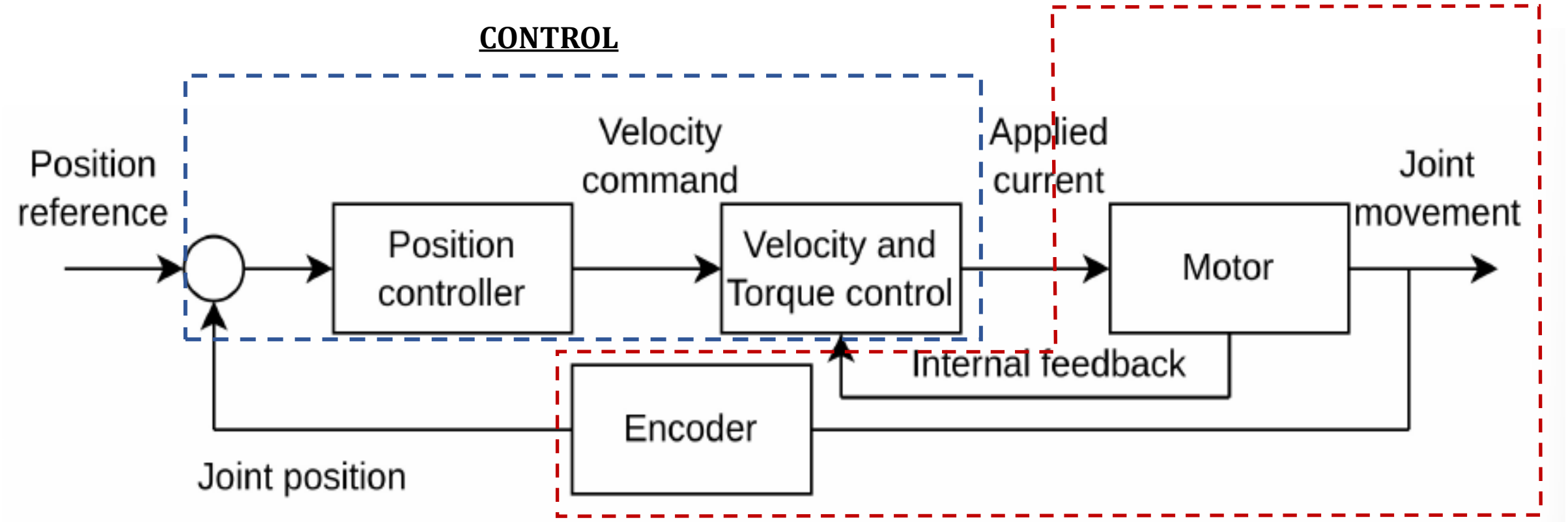
Closed Loop Control System



If you're interested take a look at: Modern Control Systems by Richard C. Dorf and Robert H. Bishop.

Closed Loop Control System

Simplest example: closed loop of a joint position control.



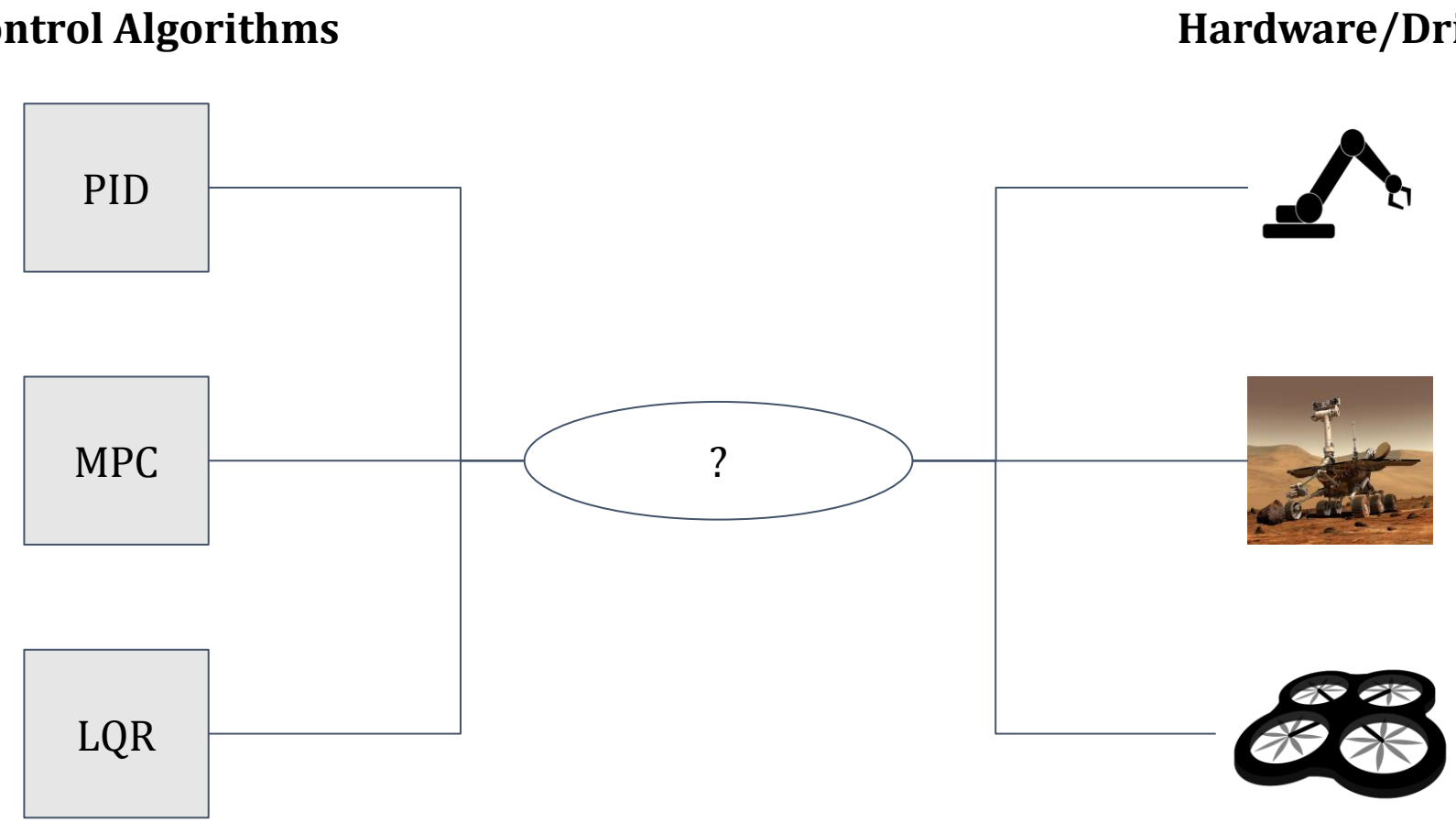
Control Algorithms from a Software Perspective

- Control algorithms are really sensitive to **delays**. They can be accounted for only if they are **known and constant**;
- **Determinism → Real-time system:**

Use real-time-grade hardware	Choose actuators, sensors, and fieldbus systems designed for **deterministic communication** to minimize unpredictable delays.
Run on a real-time OS	Use a **real-time kernel** (e.g., PREEMPT_RT Linux) and configure your controller for **real-time scheduling**. Real-time ≠ fast, it means **predictable timing**.
Synchronize data flow	Align control computation with feedback arrival and fieldbus cycles. Compute and send commands **immediately after new sensor data** is available.
Minimize inter-node latency	Prefer **shared memory** or **in-process communication**; if publishing from a real-time loop, use `realtime_tools::RealtimePublisher` instead of standard ROS publishers.
Avoid default executors	Standard ROS executors aren't deterministic. Use **custom executors** or dedicated threads for time-critical callbacks.

- **Writing real-time, deterministic software is harder but essential for fast or safety-critical control tasks**. Slow or tolerant processes can rely on nondeterministic components, high-frequency (>100 Hz) or certified controllers must run deterministically.

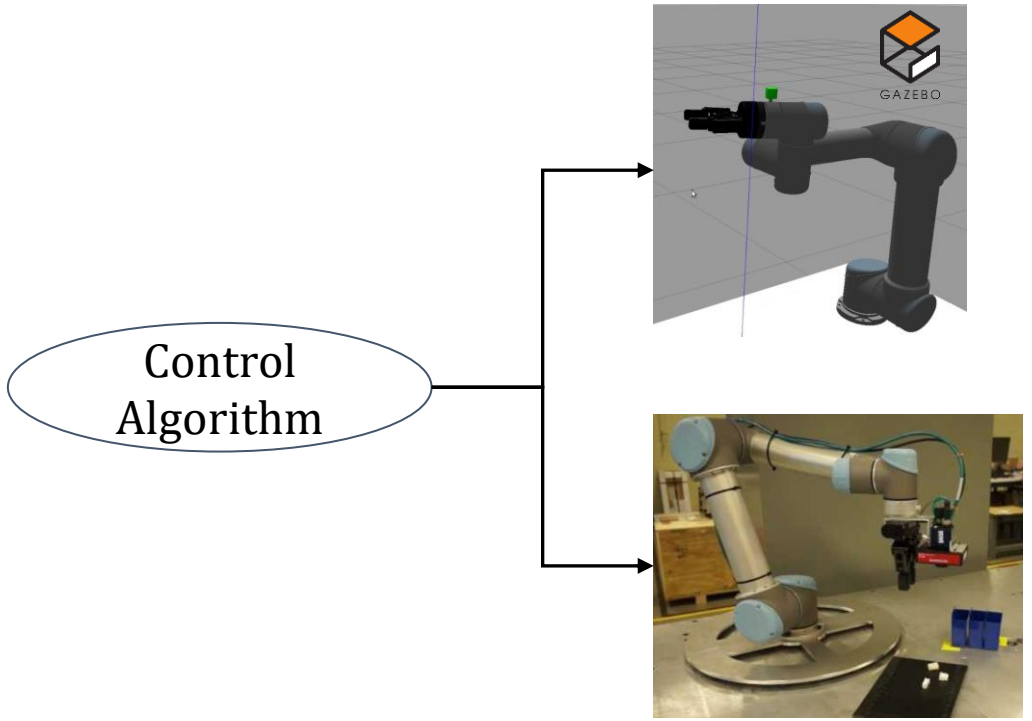
Ros2_control



Ros2_control

A **controller** computes the references for the actuators, enabling the robot motion:

- **Real-time capabilities** are required;
- Ideally, we would like to test the **same controller** in **simulation** and then, on the **real robot**, with no extra effort in adapting the code (**hardware agnostic**);
- Allows for the integration of **custom controllers**.



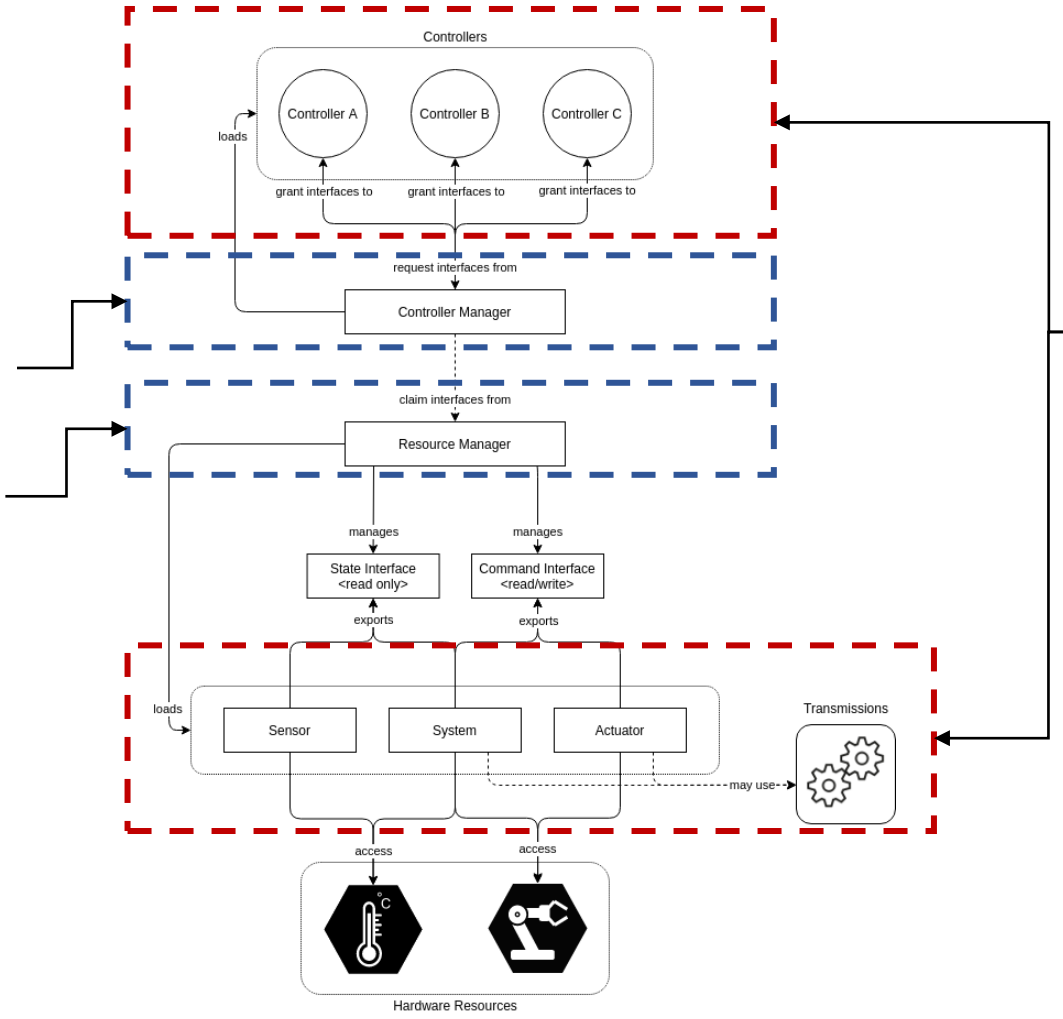
The framework ros2_control allows to do this.

Ros2_control overview

Controller Manager
- Manages controller **lifecycle**: *load, activate, deactivate, unload*
- Runs the **control loop** (*update()*): **read → update controllers → write**
- Matches required vs. provided interfaces via **Resource Manager**.

ROS 2 Node

C++ Class



Resource Manager
- Abstracts and manages **hardware components** (sensors, actuators, systems).
- Loads hardware via pluginlib.
- Provides and tracks **state** and **command interfaces.**
- Handles **read()/write()** communication during the loop.

Controllers
- Implement **control logic**.
- Derived from ControllerInterface, exported as **plugins**.
- Lifecycle managed via **LifecycleNode** (*configure → activate → deactivate*).
- Configuration of the controller manager in a YAML

C++ Plugins

Hardware Components
- Communication to physical hardware
- The components are exported as **plugins** using *pluginlib* library

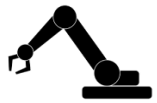
Plugins and Pluginlib

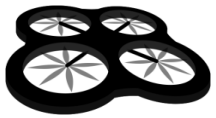
- Plugins are **dynamically loadable classes** that are loaded from a runtime library (i.e. shared object, dynamically linked library).
- Plugins are useful for **extending/modifying application behavior** without needing the application source code.

- *pluginlib* is a C++ library for loading and unloading plugins from within a ROS package: https://docs.ros.org/en/humble/Tutorials/Beginner-Client-Libraries/Pluginlib.html
- With *pluginlib*, you do not have to explicitly link your application against the library containing the classes. Instead, *pluginlib* can open a library containing exported classes at any point without the application having any prior awareness of the library or the header file containing the class definition.
- This way, you can load multiple nodes to a single process (like a ROS 1 nodelet) which enables zero-copy data transport.

Ros2_control – Hardware

Hardware/Drivers





- To communicate with the hardware, it needs to expose some **hardware interfaces**
- Usually given with the robot, otherwise you should write it
- Hardware interfaces represent the hardware through:
  - **Command Interfaces**: things that we can *control* on the robot (r/w)
  - **State Interfaces**: things that we can only *monitor* (read only)
- A robot may have *multiple* hardware interfaces

  The control manager uses a **resource manager** → loads all the interfaces together and pass them to the controller.

- How does it know about hardware interfaces?

  Configured via <ros2_control> XML section in URDF

Ros2_control – Hardware

- Connects ROS 2 Control to the **actual device or simulator**. It handles the **I/O** with the robot hardware or simulator plugin:
  - Read sensor data → publish joint states
  - Write actuator commands → motors or simulated joints

- It defines **3 hardware components:**
  - **Sensor** (state interface): hardware is used for sensing its environment. A sensor component is related to a joint (e.g., encoder) or a link (e.g., force-torque sensor).
  - **Actuator** (interfaces relevant to one joint): 1 DOF robotic hardware like motors, with reading (not mandatory) and writing capabilities. Can be used in multi-DOFs systems to communicate with each motor independently (if allowed).
  - **System** (both state and command interface): multi-DOF robotic hardware (e.g., manipulators). Can use complex transmissions like needed for robot's hands. Single communication channel to the hardware.

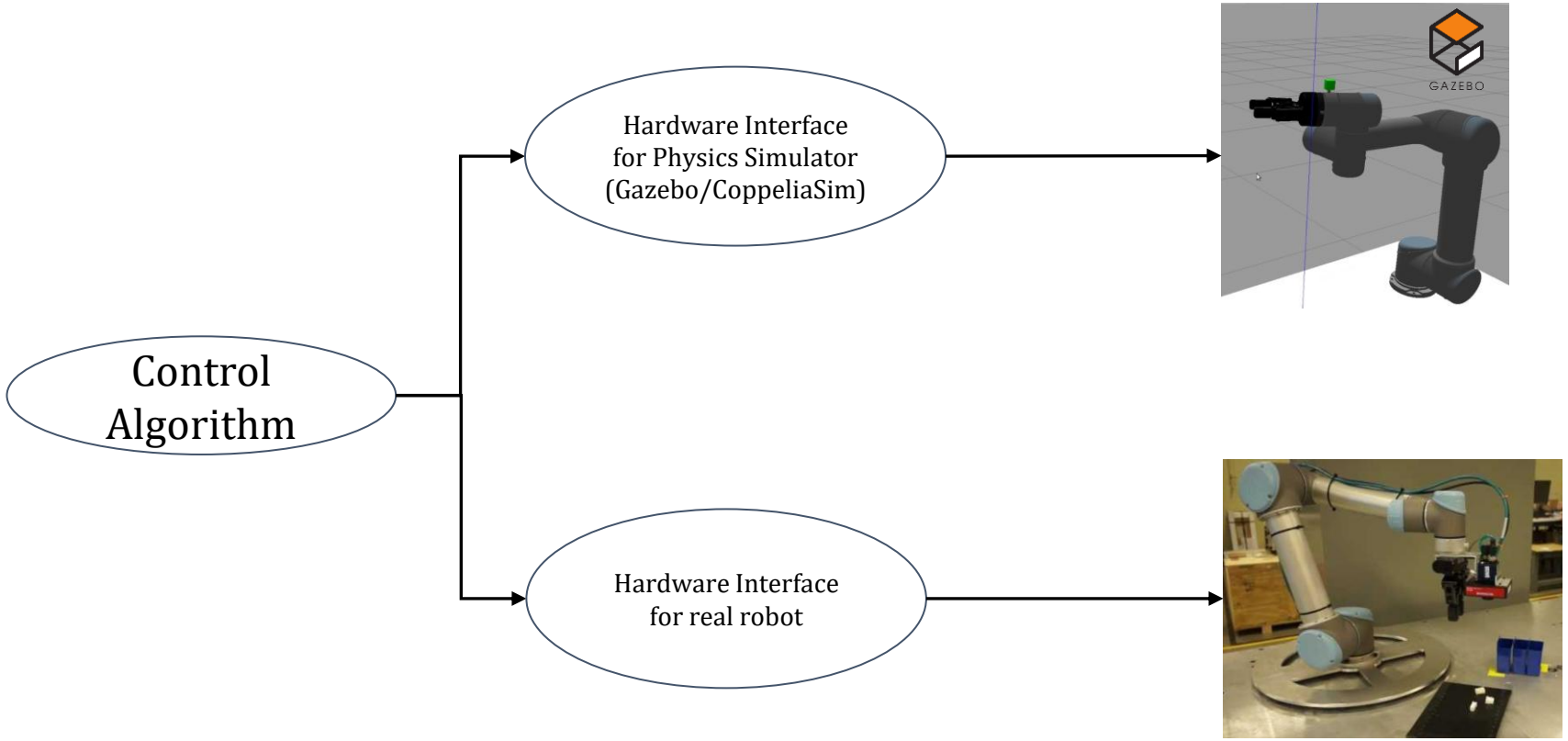
- Each category's behaviour is defined with a hardware **plugin**: `on_init()`, `on_configure()`, `read()`, `write()`.
- For each joint, the available command/state interfaces (position, velocity, effort) are declared.
- Examples of the URDF tags are available here.

Ros2_control – Hardware

```
<ros2_control name="RRBotSystemPositionOnly" type="system">
  <hardware>
    <plugin>ros2_control_demo_hardware/RRBotSystemPositionOnlyHardware</plugin>
    <param name="example_param_hw_start_duration_sec">2.0</param>
    <param name="example_param_hw_stop_duration_sec">3.0</param>
    <param name="example_param_hw_slowdown">2.0</param>
  </hardware>
  <joint name="joint1">
    <command_interface name="position">
      <param name="min">-1</param>
      <param name="max">1</param>
    </command_interface>
    <state_interface name="position"/>
  </joint>
  <joint name="joint2">
    <command_interface name="position">
      <param name="min">-1</param>
      <param name="max">1</param>
    </command_interface>
    <state_interface name="position"/>
  </joint>
</ros2_control>
```

Ros2_control – Hardware

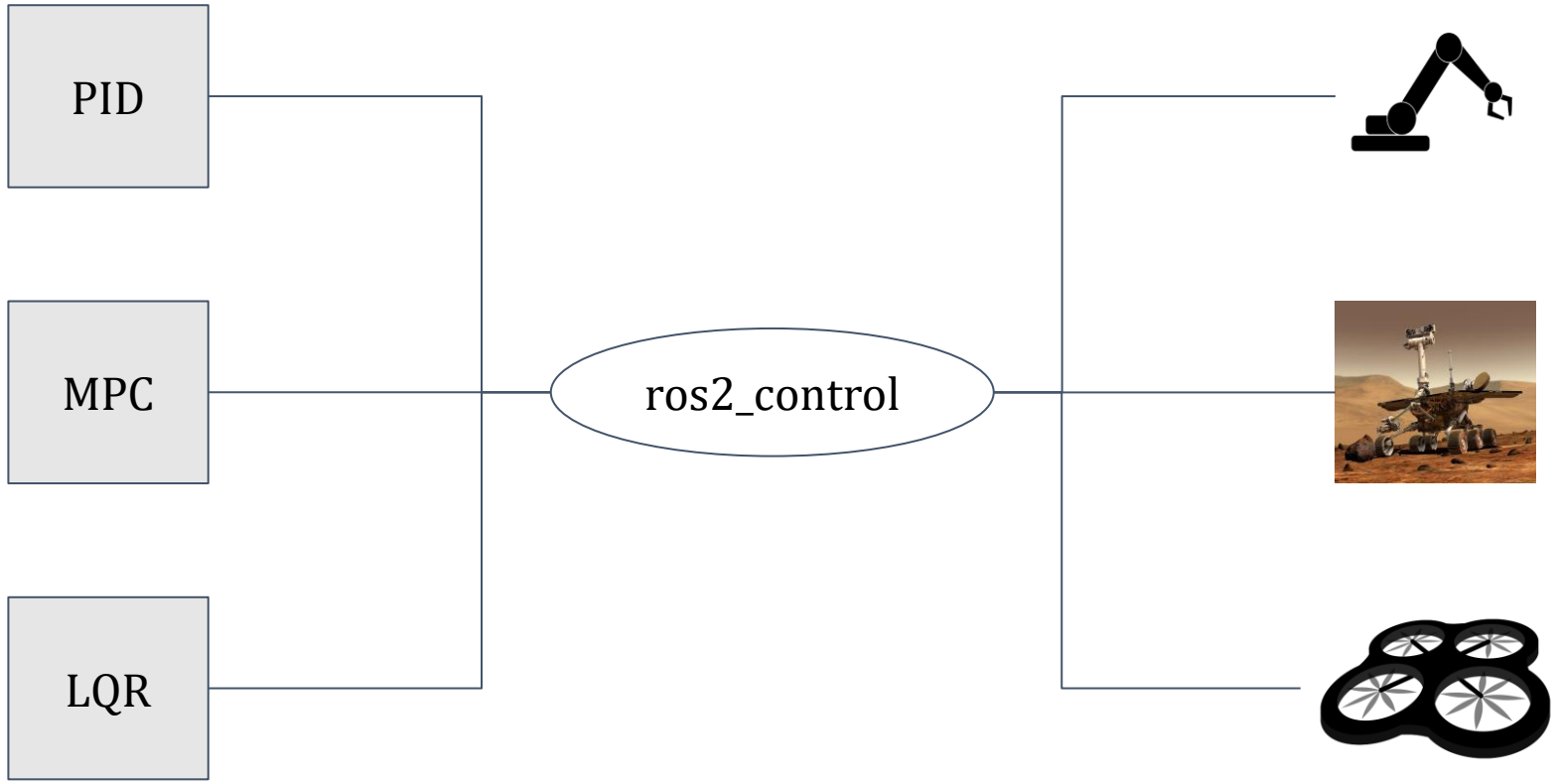
Only one hardware interface (Simulation/Real Hardware) needs to active at a time!



Ros2_control

Control Algorithms

Hardware/Drivers



Ros2_control - Controllers

Control Algorithms

PID

MPC

LQR

- They are the way ROS2 interacts with ros2_control:
  - Listen for inputs, e.g., on a topic for velocity controls
  - Calculate the correct values to pass to the hardware
  - Send the messages to the resource manager
- Controllers are designed for **specific robotics applications**
- We can have multiple controllers, as long as they drive *different command interfaces*
- How do we specify which controllers we want? Use **YAML config** files

Ros2_control – Controller Manager

- With the controller manager we can:
  - Use the provided node controller_manager/ros2_control_node
  - Spawn our own node which inherits from controller_interface::ControllerInterface for compatibility with ros2_control
- We need to provide information about
  - Hardware interfaces → URDF
  - Controllers → YAML
- To interact with the controller, we can use:
  - Services
  - CLI tools
  - Specialized nodes and scripts

Ros2_control

- Add the packages that we need to package.xml (you can use rosdep to install them)

```
<depend>ros2_control</depend>
<depend>ros2_controllers</depend>
<depend>gazebo_ros2_control</depend>
```

- Update the URDF with the ros2_control;
- Check which are the hardware interfaces available with CLI:

```
ros2 control list_hardware_interfaces
```

- Spawn the controllers:

```
ros2 control_manager spawner <controller name>
```

Ros2_control - YAML

```yaml
controller_manager:
 ros__parameters:
   update_rate: 30.0
   use_sim_time: true

   diff_cont:
     type: diff_drive_controller/DiffDriveController

   joint_broad:
     type: joint_state_broadcaster/JointStateBroadcaster

diff_cont:
 ros__parameters:
   publish_rate: 50.0
   base_frame_id: base_link
   left_wheel_names: ['chassis_to_right_wheel_joint']
   right_wheel_names: ['chassis_to_left_wheel_joint']
   wheel_separation: 0.54
   wheel_radius: 0.125
   use_stamped_vel: false
```

Ros2_control

- Spawn the controllers by running:

  ros2 control_manager spawner diff_cont

  ros2 control_manager spawner joint_broad

- Or by adding nodes to the launch file:

```
Node(
    package="controller_manager",
    executable="spawner",
    name="diff_cont_node",
    arguments=["diff_cont"]
),

Node(
    package="controller_manager",
    executable="spawner",
    name="joint_broad_node",
    arguments=["joint_broad"]
)
```

Ros2_control

- Examples with ROS 2 humble are available in the repo ros2_control_demos: https://github.com/ros-controls/ros2_control_demos/tree/humble

- Useful link (uses the previous version ROS Foxy): https://articulatedrobotics.xyz/mobile-robot-12-ros2-control/

Exercise

- Clone the UR simulation repo:
  git clone -b humble https://github.com/UniversalRobots/Universal_Robots_ROS2_GZ_Simulation.git
- Install the dependencies from binaries (rosdep install --ignore-src --from-paths src -y -r)
- If some are not found, install from source;
- Run and test the examples launch files:
  ros2 launch ur_simulation_gz ur_sim_control.launch.py
  ros2 launch ur_simulation_gz ur_sim_moveit.launch.py

  Verify the communication tree and verify packages used for simulation and control.

Use External ROS 2 Control package

- Clone the Cartesian Controllers repo in:

https://github.com/fzi-forschungszentrum-informatik/cartesian_controllers/tree/ros2

- It provides a set of Cartesian controllers, such as Cartesian position and impedance controllers.

- For human-robot interactive applications use the impedance controller
  → Can command Cartesian positions but can regulate interaction forces through stiffness variable (higher the stiffness, higher the interaction forces)

Exercise

- Use the impedance controller in Cartesian Controllers repo with simulation in CoppeliaSim (scene available at this link).