UNIVERSITÀ
DI TRENTO
Department of
Information Engineering and Computer Science

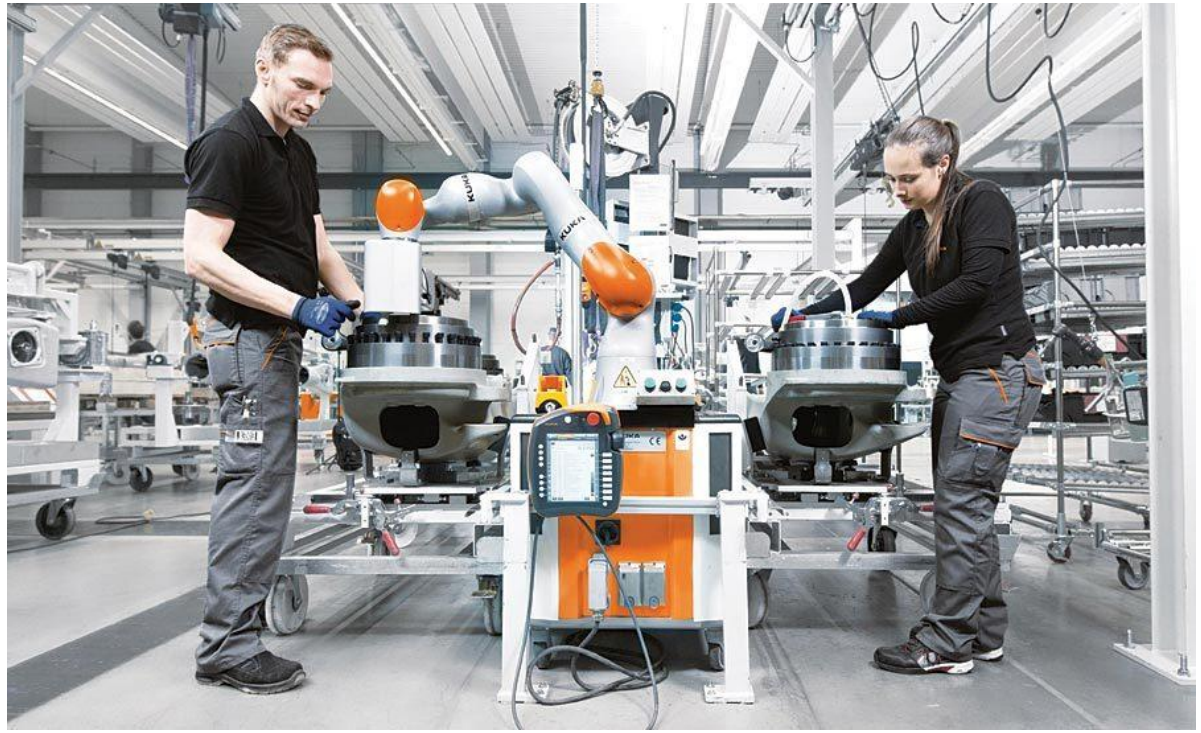
Robot Programming

Edoardo Lamon

Software Development for Collaborative Robotics

Academic Year 2025/26

Industrial vs Collaborative Robotics



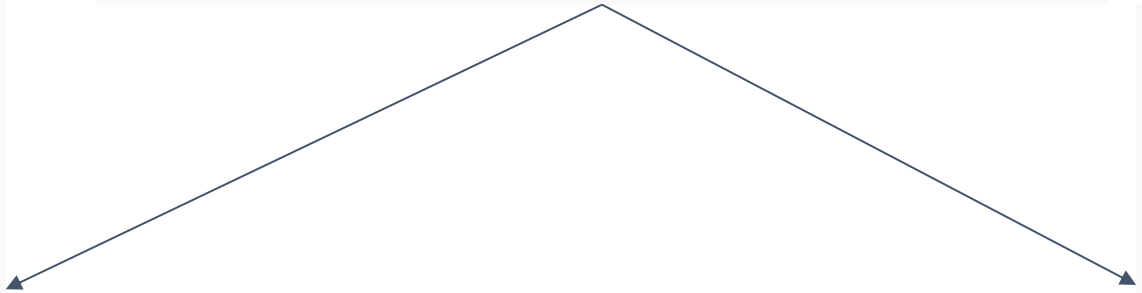
The World of Industrial Robotics

- "**Closed**" environment to execute repetitive tasks
- Robot programming can be done in two ways
  - **Guiding**: the robot arm is guided (manually or by a remote controller) through a set of points
  - **Off-line**: the robot follows a program written in a script language

- The script languages used for programming are often **proprietary**
  - ….. But extremely simple and similar with each other

Example – Offline Programming

```
Move to P1 (a general safe position)
Move to P2 (an approach to P3)
Move to P3 (a position to pick the object)
Close gripper
Move to P4 (an approach to P5)
Move to P5 (a position to place the object)
Open gripper
Move to P1 and finish
```



```
Function PickPlace
        Jump P1
        Jump P2
        Jump P3
        On vacuum
        Wait .1
        Jump P4
        Jump P5
        Off vacuum
        Wait .1
        Jump P1
Fend
```

```
PROGRAM PICKPLACE
        1. MOVE P1
        2. MOVE P2
        3. MOVE P3
        4. CLOSEI 0.00
        5. MOVE P4
        6. MOVE P5
        7. OPENI 0.00
        8. MOVE P1
.END
```

Offline Languages

Robot brand	Language name
ABB	RAPID
Comau	PDL2
Fanuc	Karel
Kawasaki	AS
Kuka	KRL
Stäubli	VAL3
Yaskawa	Inform

The scripts are interpreted and translated on the fly into real-time actions

Modern Robots

- Modern robots are much more complex:
  - Open environments;
  - Perception abilities;
  - Re-plan in real-time;
  - Human interaction;
  - Robot collaboration.

- Each discipline has its own programming framework and languages.
- So integration can be a titanic effort.

A Quick (and Incomplete) Survey

Activity	Methodologies	Framework → Languages
Sensing and actuation	Micro-controller programming	FreeRTOS, proprietary IDE → C
Kinematic and dynamic control	Model based control design	MATLAB-Simulink → C/C++
Perception (detection/classification)	Machine learning	Yolo, OpenCV → Python, C/C++
SLAM, data fusion	Statistical learning	MATLAB-Simulink → C/C++
Motion planning	Optimisation techniques	MATLAB-Simulink, libraries → C++
Task planning	Discrete optimization, formal methods	PDDL, … → Python, C++

Additional Problems

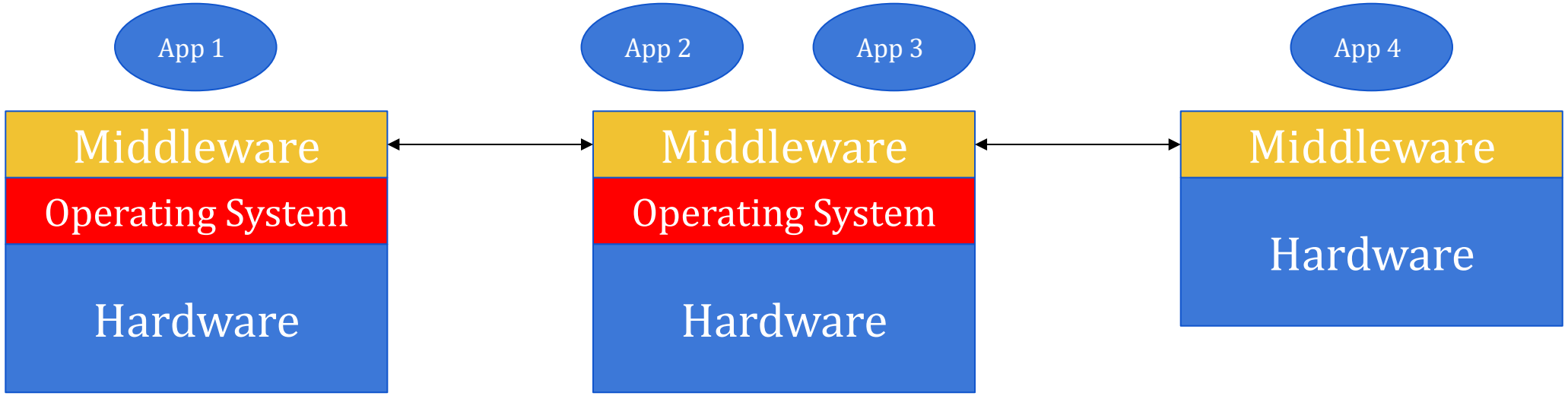
- **Robot-to-robot** communication
- External services in the **cloud** (e.g., for strategic decisions)
- **Heterogeneous** types of hardware
  - Microcontrollers
  - GPU
  - Industrial PC
- **Timing constraints** on computations
  - Particularly true for unstable systems like drones or legged robots
- Finally, most of the times the developers of SW components for robotics are not exactly computer experts …not your case ☺

The Solution

Middleware: A computer software that enables communication between multiple software applications, possibly running on more than one machine.

Development of *distributed*, *multilingual* applications without requiring the direct use of Operating System and networking primitives

The Concept of Middleware



- **Standard messages**
- Abstraction w.r.t. applications position
- The middleware sees to the **correct delivery** of the messages
- Applications can be written in **any language** as long as they get connected through a *client library*
- Applications are **OS independent** (which in some case is not there)

The Concept of Middleware

- Middleware is an abstraction layer that significantly simplifies the development and the integration of distributed applications.
- There are many types of middlewares:

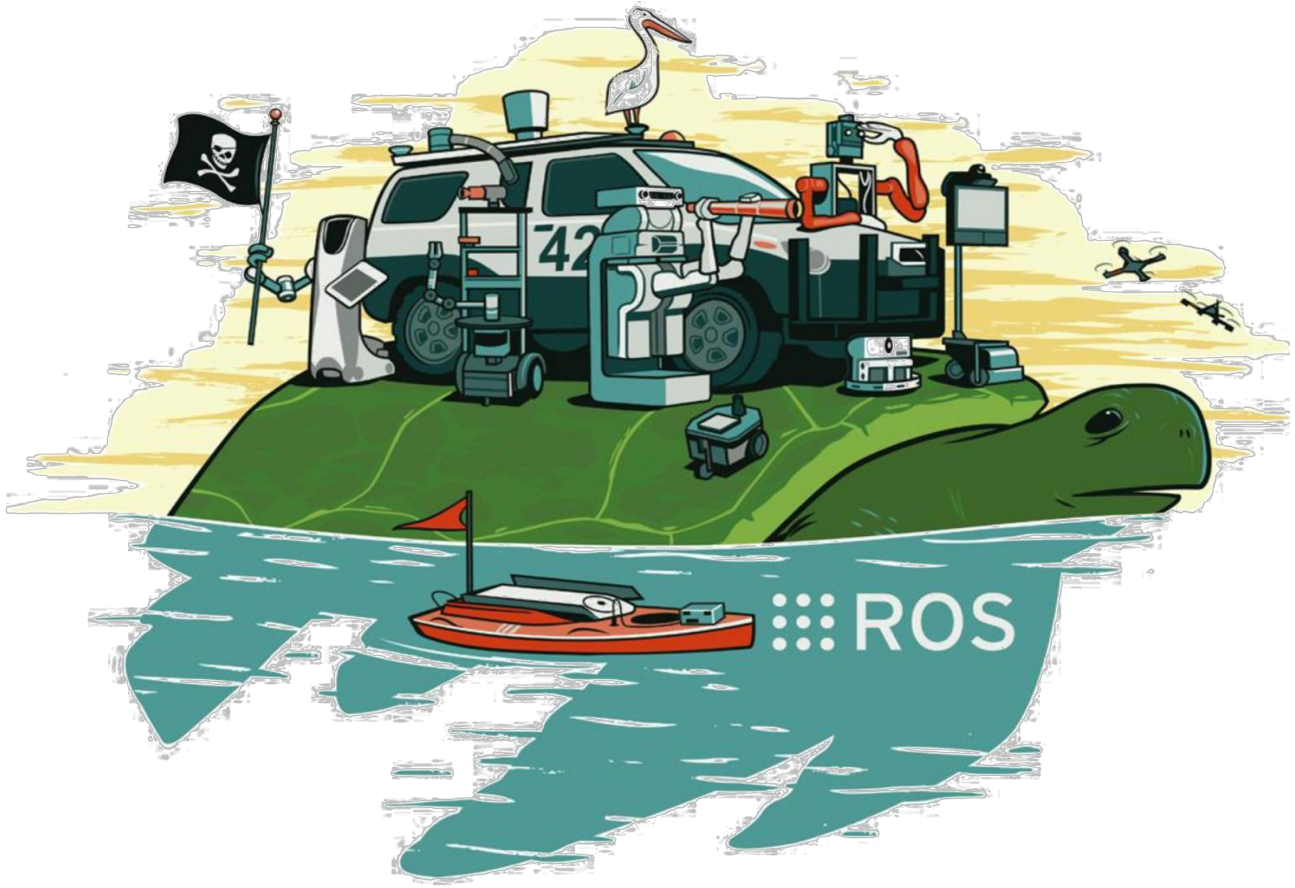
Type	Services	Examples
Message Oriented Middleware	Receiving and sending of messages over distributed applications	Amazon Simple Notification System (SNS), IBM MQ, Amazon AWS IoT Core
Remote Procedure Call (RPC)	Calling procedures on remote systems and performing synchronous or asynchronous interactions	Oracle: Open Network Computing RPC, SOAP
Database Middleware	Allowing for direct access to databases	ODBC, JDBC, EDA/SQL
Embedded Middleware	Supporting embedded applications	zMQ, ROS, IoT middlewares

Middlewares for Robotics

- **Robot Operating System (ROS)**
  - De-facto standard for hundreds of available components
  - Some issues with complexity and latency… (addressed by ROS 2)
- **zMQ**
  - Message oriented middleware for lightweight embedded applications
  - Usable (and used) in robotics
  - Very low and controlled latencies
  - Integrated into ROS 2

In this course, we will use ROS 2 for the huge availability of software and services!

ROS 2

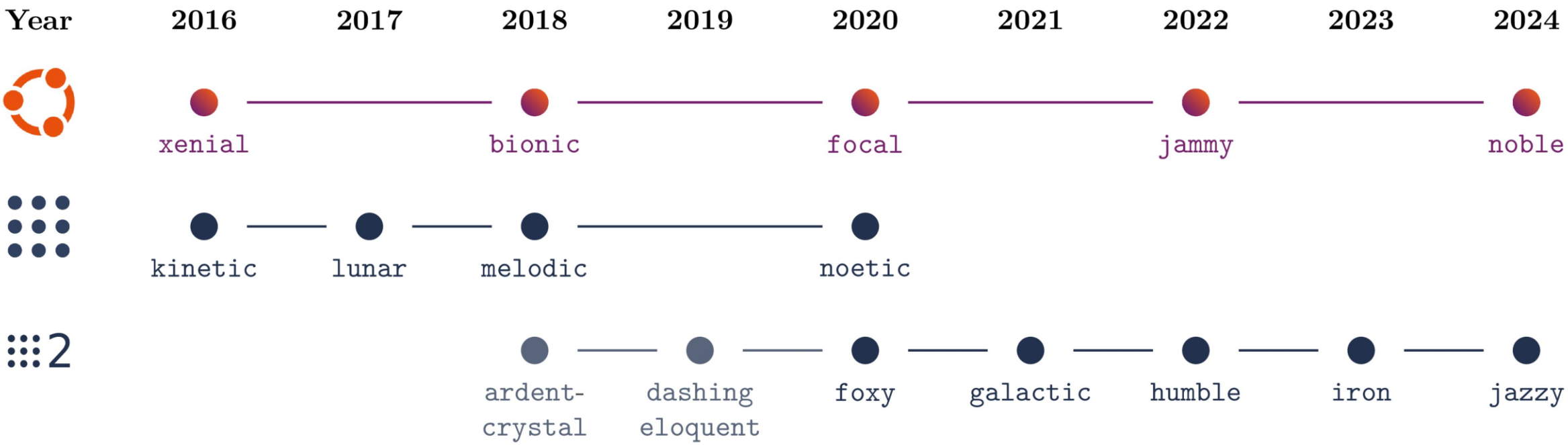


(Image taken from Willow Garage's "What is ROS?" presentation)

History of ROS

- ROS was initially developed in Stanford (Artificial Intelligence Laboratory) in 2007
- Since 2013 the project was taken over by OSRF (**Open Source Robotics Foundations**)
- It has become a de-facto standard, very popular especially (but not exclusively) at the academic level
- Growing industrial success and recognition (ROS Industrial)
- Some robot and sensors manufacturers provide ROS integration

ROS Timeline



ROS 2 Distributions

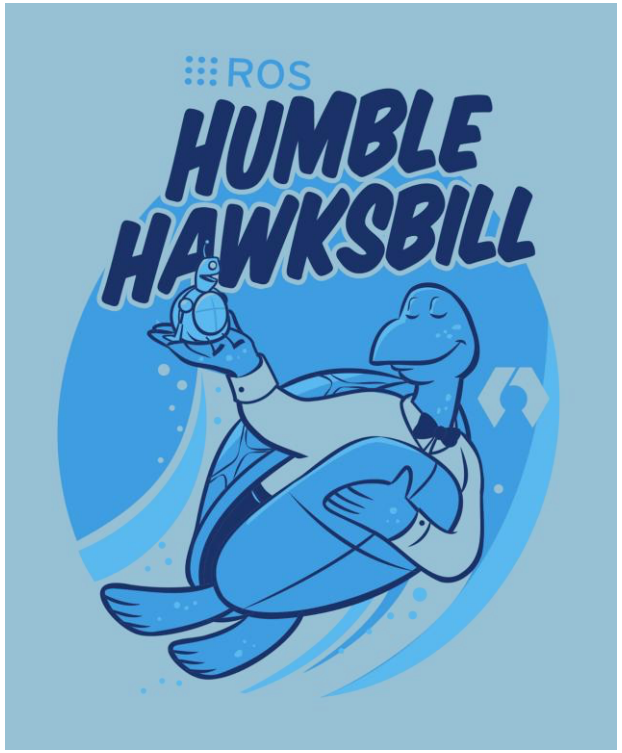
The latest ROS 2 release is “Kilted”, but we will work with version “**Humble**" (along Ubuntu 22.04)

Logo	ROS2 Distro	Release date	Ubuntu Distro	EOL date
	Kilted Kaiju	May 2025	Ubuntu 24.04	December 2026
	Jazzy Jalisco	May 2024	Ubuntu 24.04	May 2029 (**LTS**)
	Iron Irwini	May 2023	Ubuntu 22.04	December 2024
	Humble Hawksbill	May 2022	Ubuntu 22.04	May 2027 (**LTS**)

ROS 2 Rolling Ridley is the rolling development distribution of ROS 2. Rolling is continuously updated and **can have in-place updates that include breaking changes**. So do not use it (unless you’re a ROS2 developer).

Main Differences: ROS1 – ROS2

Difference	ROS1	ROS2
Communication Middleware	TCP/UDP	Data Distribution Service (DDS)
Real-Time	No support	Supports safety-critical systems
Multi-Thread Execution	No support	Multiple running nodes
Language Support	Python and C++	Python, C++, Rust, Java, C# and JavaScript
OS Support	Ubuntu	OS-agnostic
Security	Low	Incorporates authentications, encryption, etc.



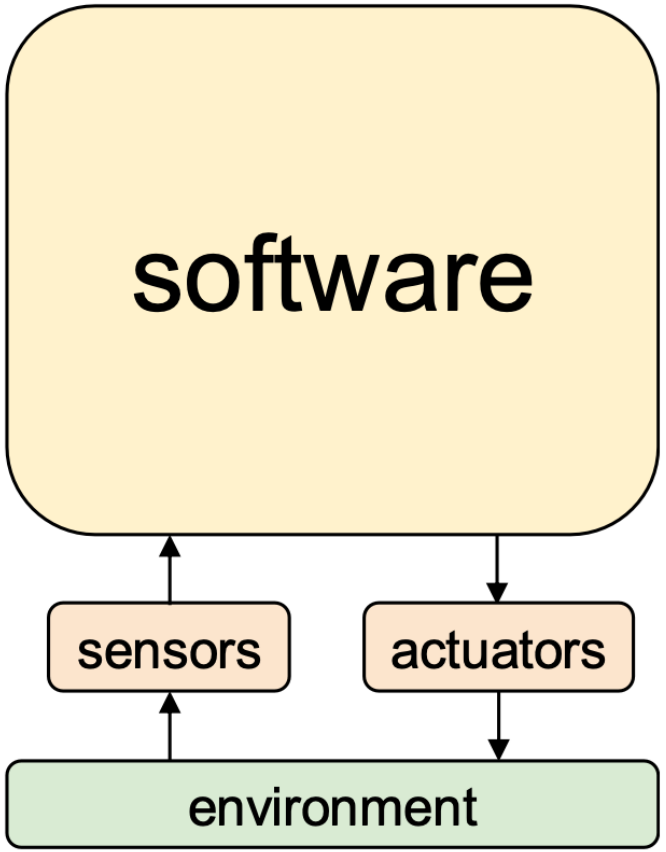
Transition to ROS 2

- Community is currently in transition!
  - Final ROS1 release (Noetic) is out (EOL in 2025)
  - All critical features are now supported in ROS 2
- ROS Industrial will take time to transition
  - Many breaking changes / conceptual differences
  - Goal: industrial robots will become native ROS devices

What is a Robot?

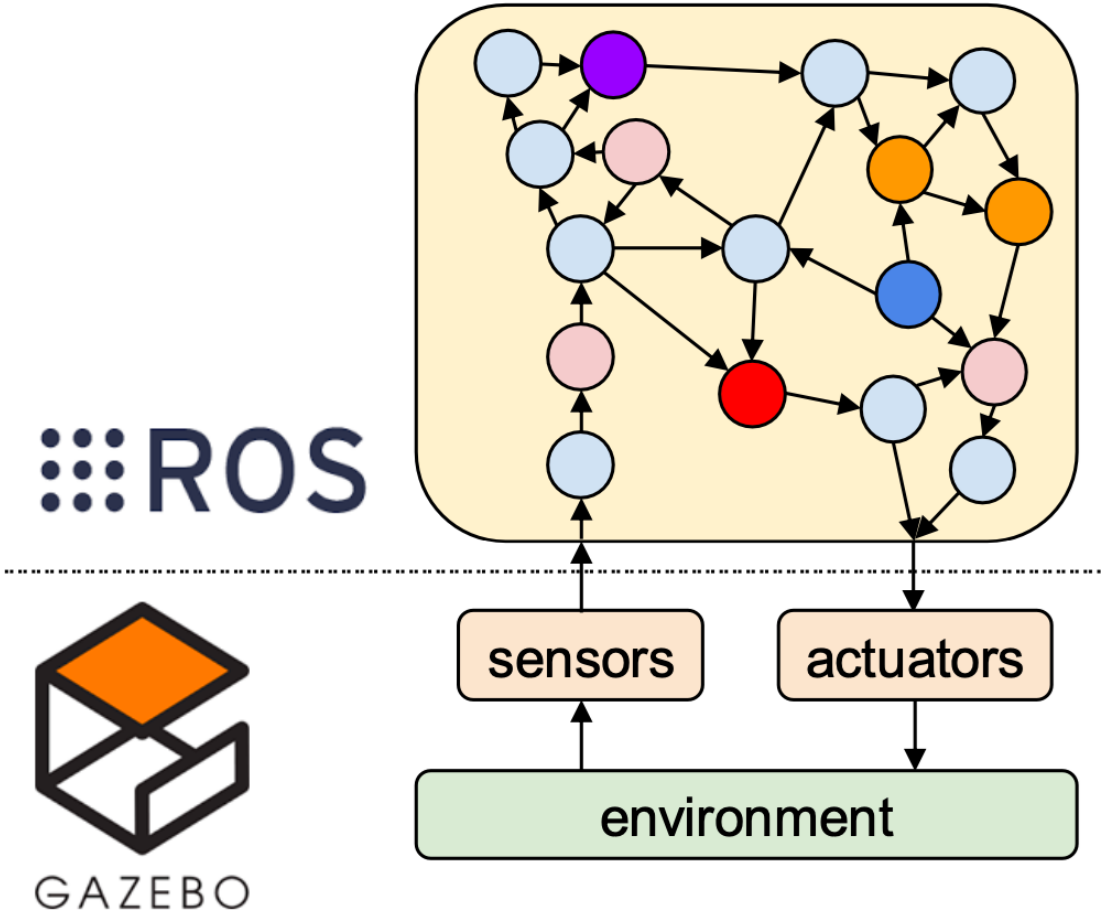
Software connecting sensors to actuators to interact with the environment

(Adapted from Morgan Quigley's "ROS: An Open-Source Framework for Modern Robotics" presentation)



How Can ROS 2 Help?

- Break Complex Software into **Smaller Pieces**
- Provide a framework, tools, and interfaces for **distributed development**
- Encourage **re-use** of software pieces
- Easy transition between simulation and hardware



Main ROS 2 Features

- **Peer-to-peer**: applications use a standardized API to exchange messages
- **Distributed**: the framework fully supports applications running on multiple computers
- **Multilingual**: the ROS components can be developed in any language as long as a client library exists (C++, Python, Matlab, Java, Ruby…)
- **Light-weight** (especially 2.0): applications are connected through a very simple and thin layer
- **Free and open-source**: most of ROS applications are open-source and free to use. But, its permissive licensing policy allows for the development of closed and commercial applications

Get Started with ROS 2

Prerequisites: Ubuntu 22.04 LTS

Options:

- Native: Dual Boot (**RECOMMENDED**)
- Virtual: using apps like Windows Subsystem for Linux (WSL) , Docker (e.g. for Mac users) etc.

INSTALLATION of Ubunt 22.04 in WSL
In Windows 10-11, open a powershell prompt (Terminal) and run:

```
wsl --install -d Ubuntu-22.04
sudo apt update; sudo apt full-upgrade
sudo apt install x11-apps --yes
```

See also this guide for more info about WSL.

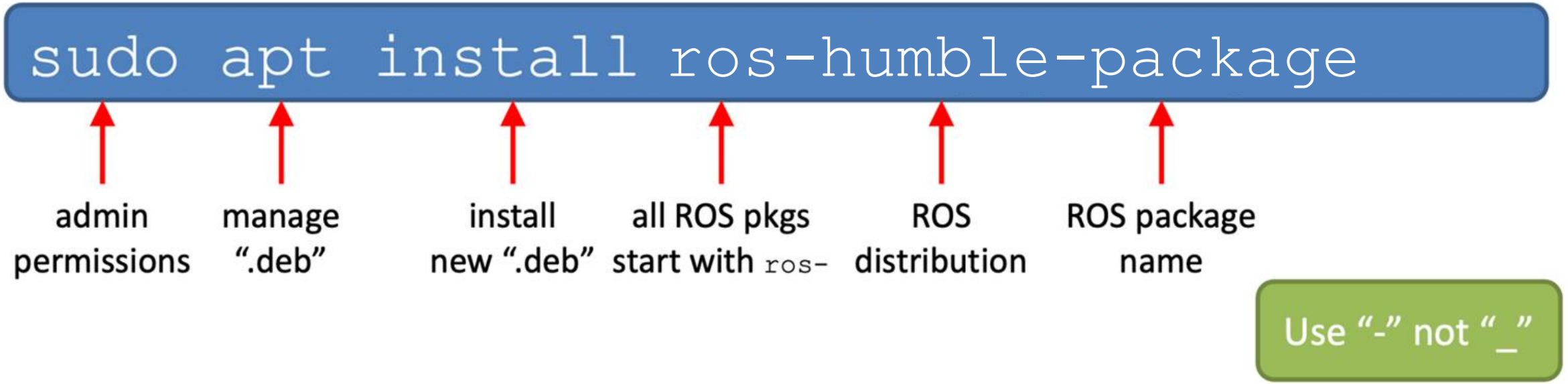
ROS 2 Humble Installation

We will perform a desktop installation of ROS 2 Humble (in Ubuntu 22.04):

```
sudo apt install software-properties-common
sudo add-apt-repository universe

sudo apt update && sudo apt install curl -y

export ROS_APT_SOURCE_VERSION=$(curl -s https://api.github.com/repos/ros-infrastructure/ros-apt-source/releases/latest | grep -F "tag_name" | awk -F\" '{print $4}')

curl -L -o /tmp/ros2-apt-source.deb "https://github.com/ros-infrastructure/ros-apt-source/releases/download/${ROS_APT_SOURCE_VERSION}/ros2-apt-source_${ROS_APT_SOURCE_VERSION}.$(. /etc/os-release && echo ${UBUNTU_CODENAME:-${VERSION_CODENAME}})_all.deb"

sudo dpkg -i /tmp/ros2-apt-source.deb

sudo apt update; sudo apt upgrade

sudo apt install ros-humble-desktop

source /opt/ros/humble/setup.bash
```

- See also this guide for installation details.

ROS 2 Packages Installation from deb



ROS 2 Concepts

Nodes, Topics, Services, Actions, Parameters

ROS 2 Nodes

- A node is a *single process* delivering a service (c++ or python program)
- Nodes can be combined together to form **graphs**
- Each node in ROS should be responsible for a single, module purpose. For example:
  - One node manages the RGBD camera
  - One node implements skeletal tracking
  - One node implements motion planning
  - One node implements Cartesian control
- The use of nodes allows the developers to *decouple their work* and improve **maintainability** and **robustness** of their code

ROS 2 Nodes

- Nodes uses a client library to communicate with other nodes:
  - rclcpp - ROS Client Library for C++
  - rclpy - ROS Client Library for Python (converts messages in plain C, so it is slower)
  - additional client libraries (C, JVM, C#, Rust, etc.) are maintained by ROS community
- Client libraries expose to users the core ROS functionalities:
  - Names and namespaces
  - Time (real or simulated)
  - Parameters
  - Console logging
  - Threading model
  - Intra-process communication
- ROS Client Library (RCL) interface implements logic and behavior of ROS concepts is not language-specific. However, common RCL functionality is exposed with C interfaces that are easy to wrap with a client library
- To program our custom nodes, we will use the client libraries (mainly rclcpp).

ROS 2 Command Line Tools

- ROS 2 includes a suite of **command-line tools** for introspecting a ROS 2 system: ros2cli

- **Usage:** The main entry point for the tools is the command ros2, plus various sub-commands (run ros2 --help in the terminal to check them)

- Since ROS 2 uses a distributed discovery process, it can take time to discover all nodes

- When we use cli tools a new inspection starts in background (using the ROS 2 daemon)

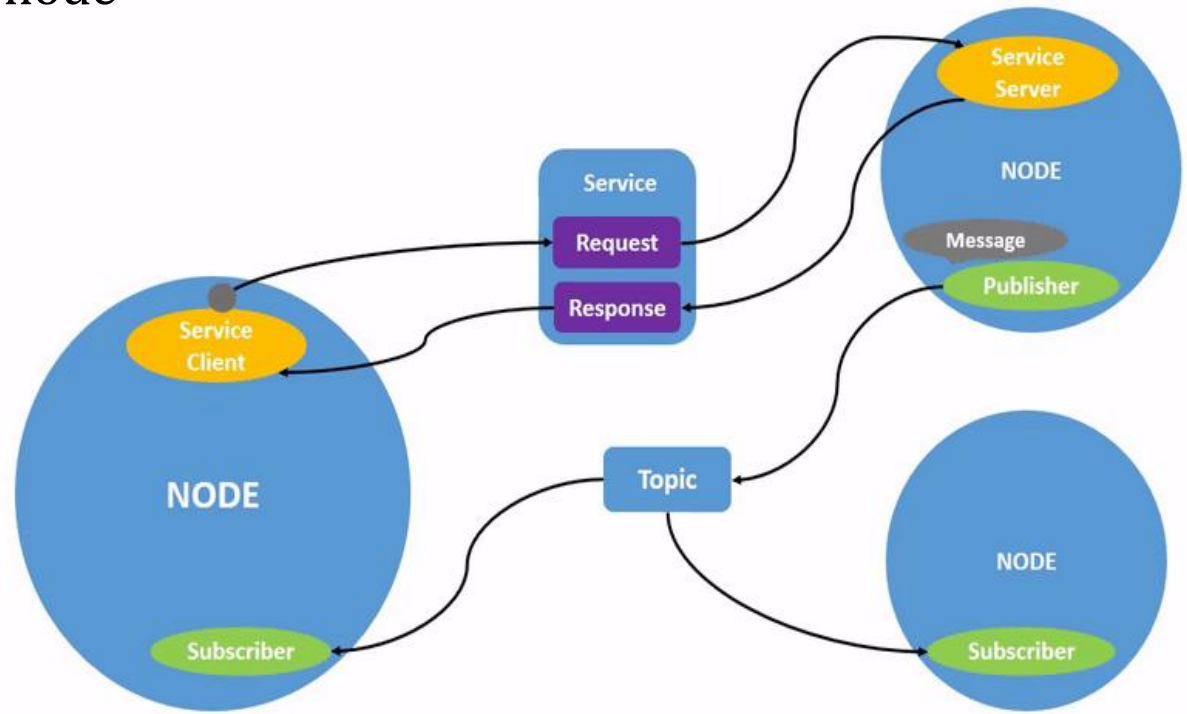
ROS 2 Nodes

Let's inspect nodes using the ROS 2 command-line tools:

- They are organised in packages. Execution of a node
  $ ros2 run <package_name> <node_name>

- List of active nodes
  $ ros2 node list

- Information retrieval on a node
  $ ros2 node info <node_name>



Communication between nodes

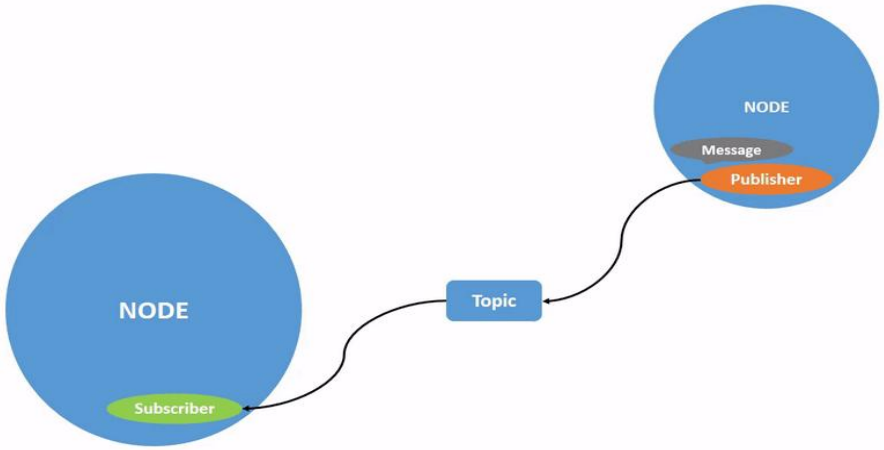
- There are four ways to communicate between nodes in ROS2
  - **Streaming topics**
  - **Remote Procedure Call (RPC) Services**
  - **Actions**

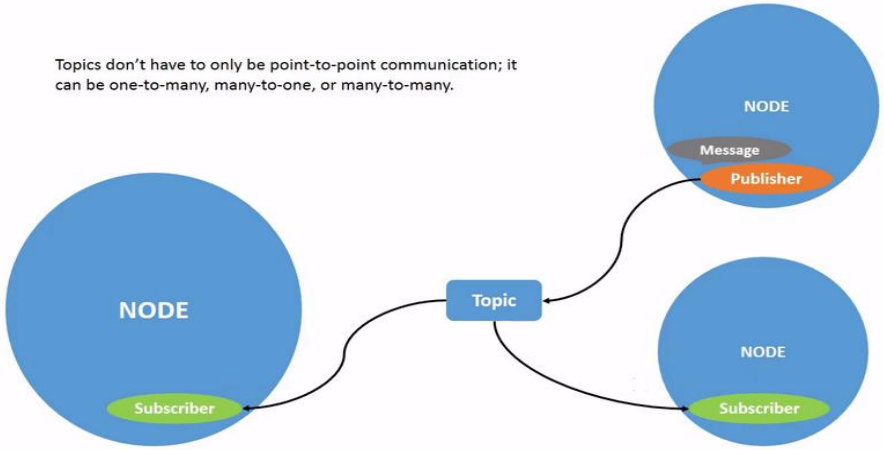
- In addition, to configure nodes at startup (and during runtime) without changing the code, the **parameter server** is used

- Let us focus on topics and messages

Topics and Messages

- A topic is a name for a stream of messages
- Topics are the primary way for establishing a communication
- Nodes can **publish** or **subscribe** to a topic
- This scheme is intended for **unidirectional streaming**
- Abstraction between subscriber-publisher






```
$ ros2 topic list
```

Messages

- Nodes communicate through topics *exchanging messages*
- A message defines the type of a topic
- It is defined in a **.msg** file
- It is possible to create **custom** .msg
- A message is a data structure of (full list):
  - Fields
  - Integers
  - Booleans
  - Strings
  - Structs (c-like)
- ROS 2 provides several packages of messages: std_msgs, geometry_msgs, nav_msgs, etc. (use them first!)

geometry_msgs/Point.msg

```
float64 x
float64 y
float64 z
```

sensor_msgs/Image.msg

```
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
uint32 height
uint32 width
string encoding
uint8 is_bigendian
uint32 step
uint8[] data
```

Messages

- Messages can be used as field in other messages:

geometry_msgs/Point.msg

```
float64 x
float64 y
float64 z
```

sensor_msgs/Image.msg

```
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
uint32 height
uint32 width
string encoding
uint8 is_bigendian
uint32 step
uint8[] data
```

geometry_msgs/PoseStamped.msg

```
std_msgs/Header header
 uint32 seq
 time stamp
 string frame_id
geometry_msgs/Pose pose
  geometry_msgs/Point position
    float64 x
    float64 y
    float64 z
  geometry_msgs/Quaternion orientation
    float64 x
    float64 y
    float64 z
    float64 w
```

Messages

- The type of a topic (i.e., the structure of the messages exchanged) can be seen by

  $ ros2 topic type /topic

- A message can be published by

  $ ros2 topic pub /topic type data

- Naming convention: package+name of the .msg file

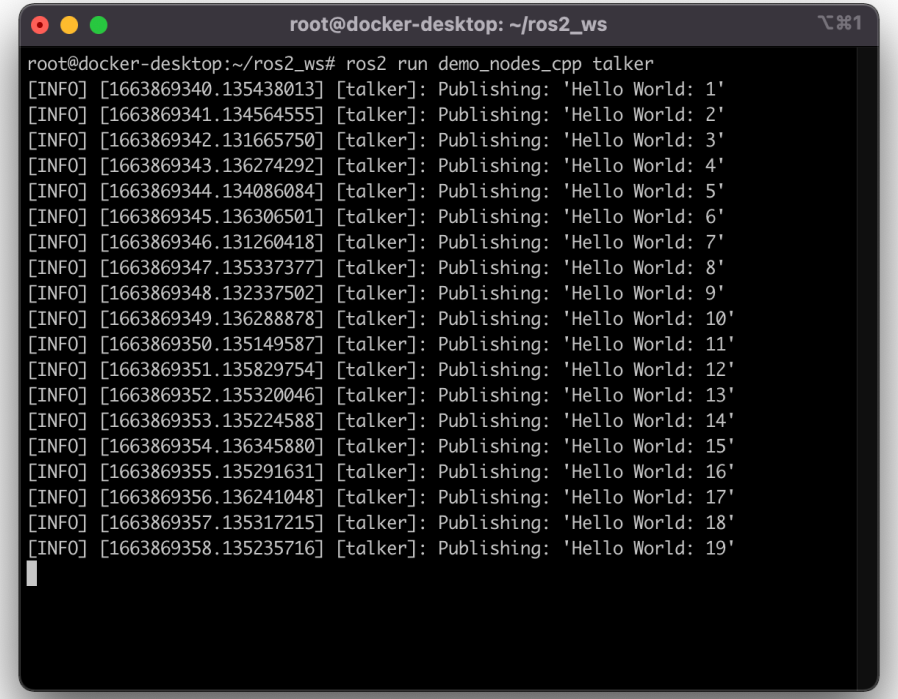
  std_msgs/msg/String.msg

Node Graphs

- The ROS graph is a network of ROS 2 elements processing data together at one time
- It encompasses all executables and the connections between them if you were to map them all out and visualise them.
- One way to visualize them is using rqt_graph
- But at the moment you will not see anything, as no node is running. Let's try an example.

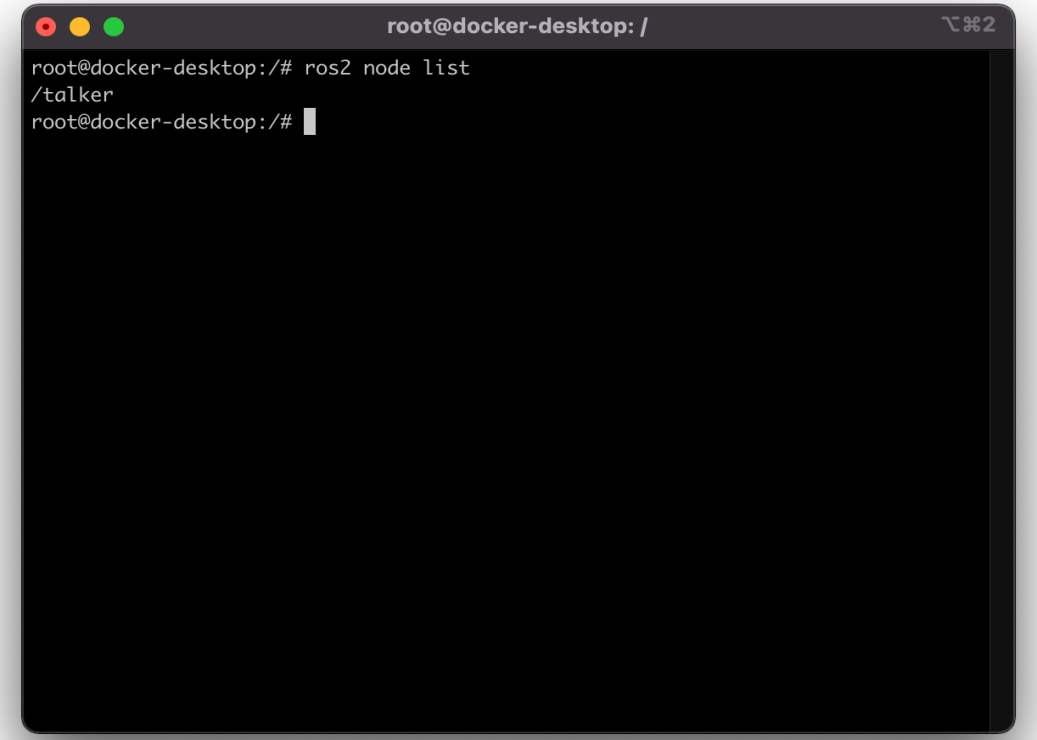
Example – Nodes & Topics

$ ros2 run demo_nodes_cpp talker



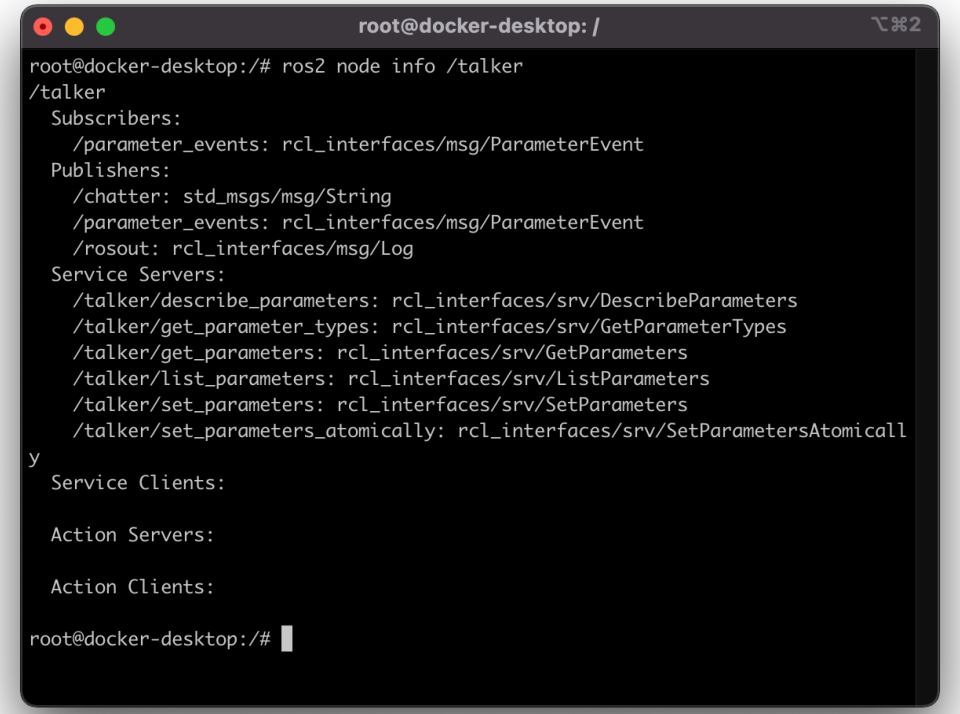
Example – Nodes & Topics

$ ros2 node list



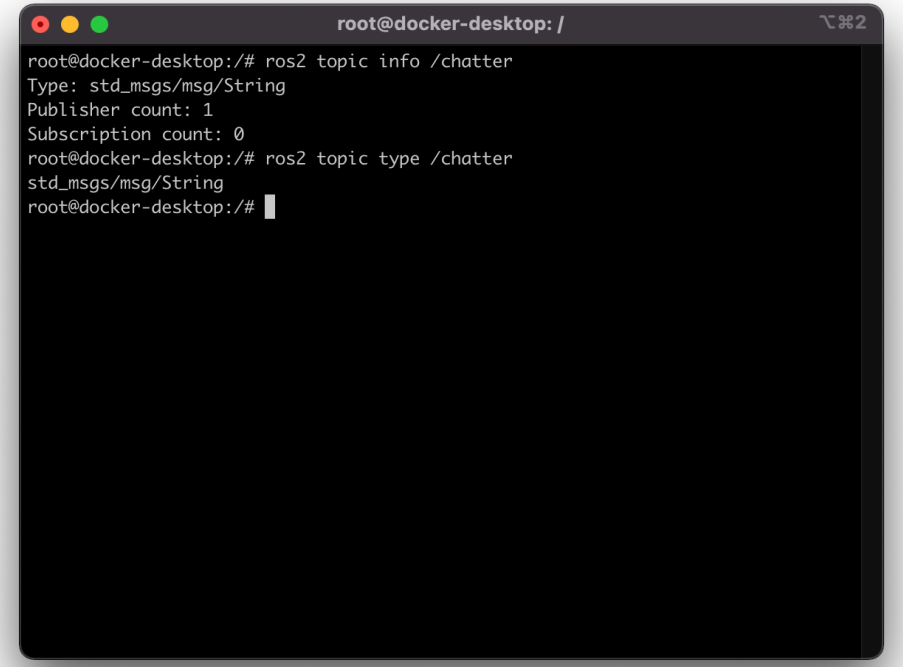
Example – Nodes & Topics

$ ros2 node info /talker



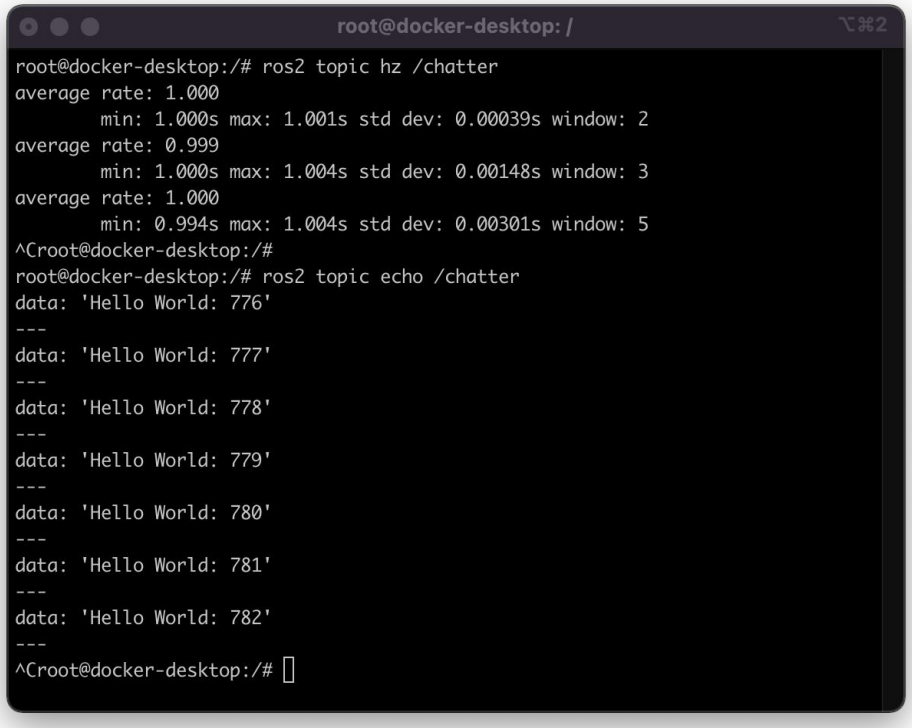
Example – Nodes & Topics

$ ros2 topic info [--verbose] /chatter
$ ros2 topic type /chatter



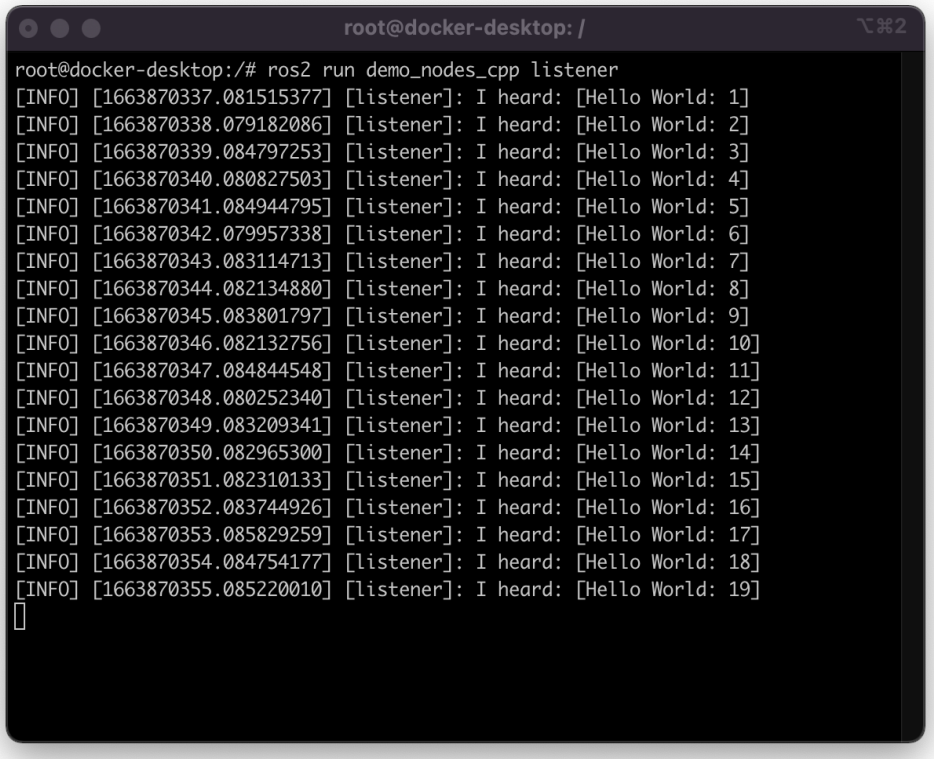
Example – Nodes & Topics

$ ros2 topic hz /chatter
$ ros2 topic echo /chatter



Example – Nodes & Topics

$ ros2 run demo_nodes_cpp listener

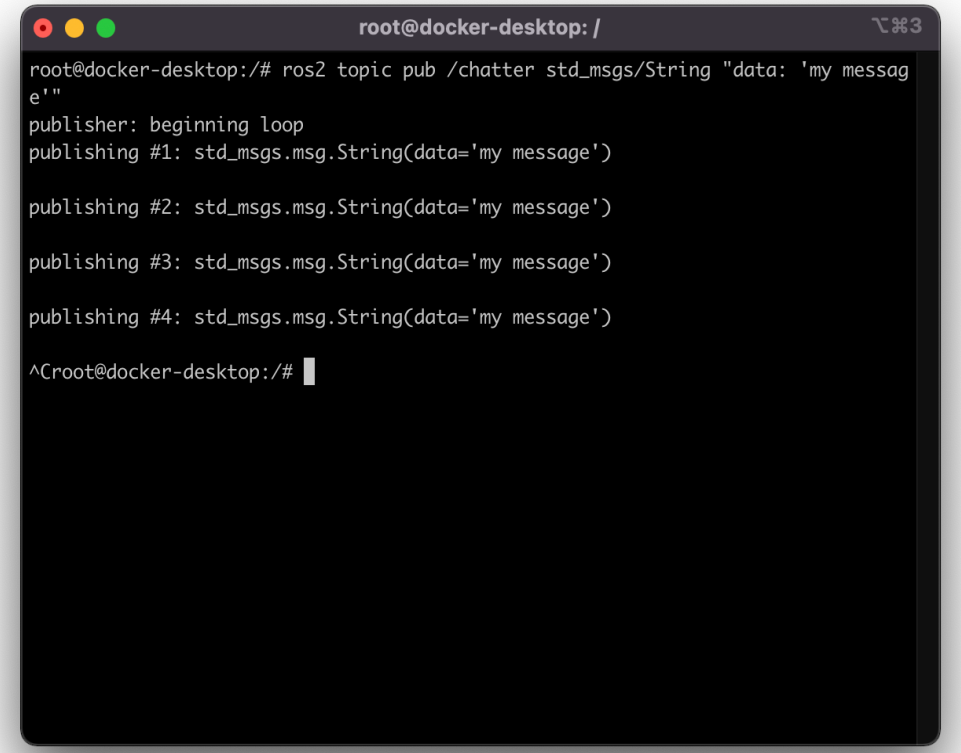


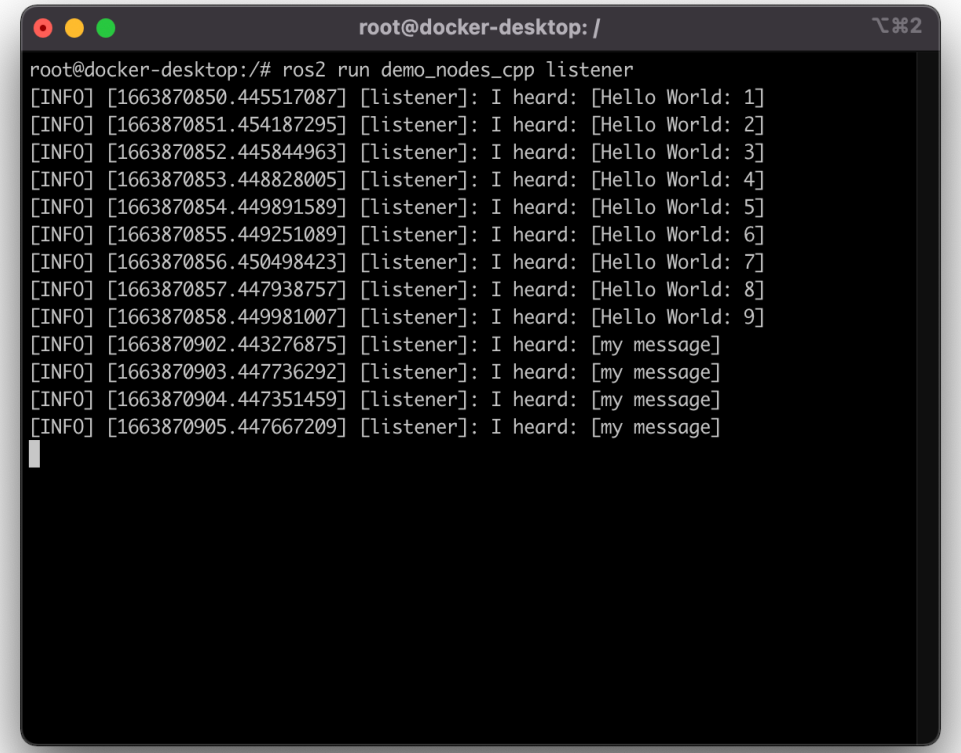
Example – Nodes & Topics

$ ros2 node list
$ ros2 topic info [--verbose] /chatter

Example – Nodes & Topics

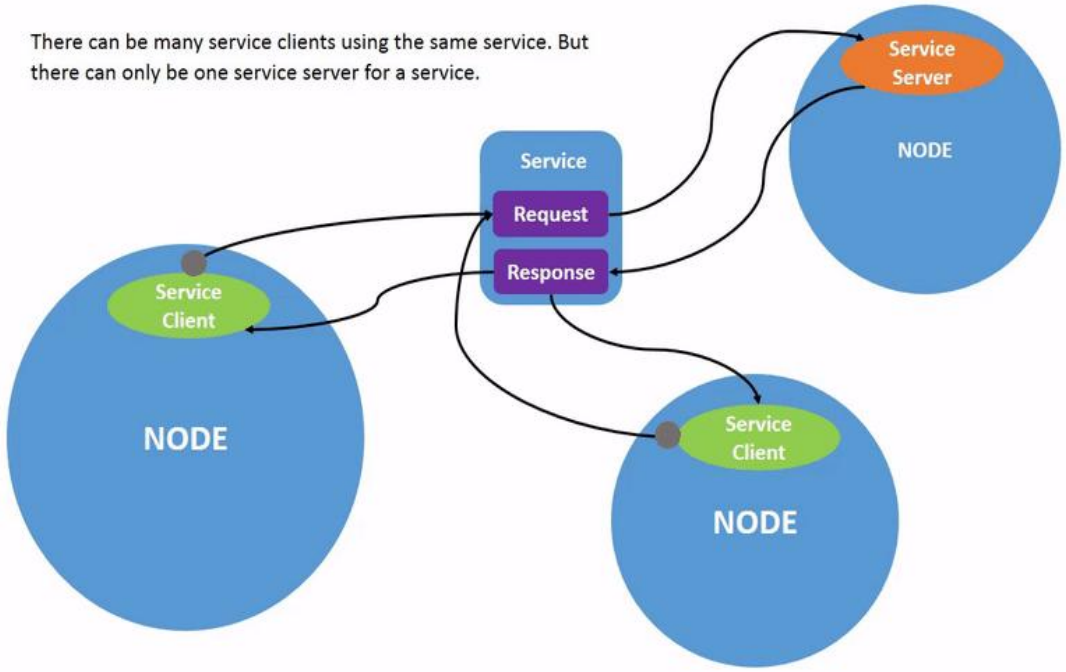
$ ros2 topic pub /chatter std_msgs/String "data: 'my message'"
$ ros2 topic pub [--once] [-t times] [-r rate] ...





ROS 2 Services

- Services are based on a **call-and-response** model, versus topics' publisher-subscriber model
- Service refers to a **remote procedure call**.
- One or multiple nodes (**service clients**) can make a remote procedure call to another node (**service server**) which will do a computation and return a result.
- Services are expected to return quickly, as the client is generally waiting on the result.



ROS 2 Services

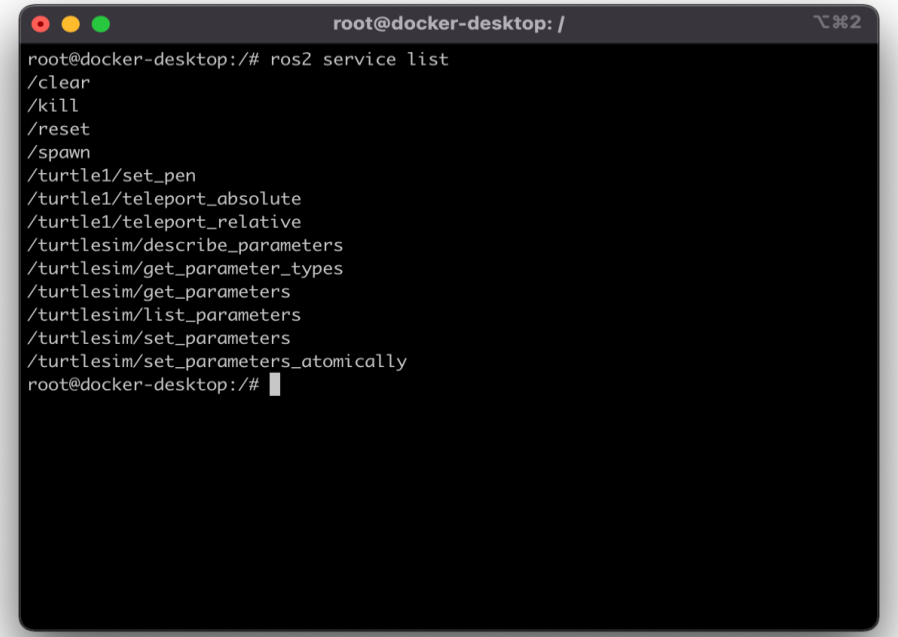
- Also services have a specific interface type, like messages for topics
- The service interface is defined in a **.srv** file (same data structures supported as .msg)
- It is possible to create **custom** .srv
- Differently from messages, ROS 2 provides only one “standard” package for services std_srvs, which contains:
  - Empty.srv: A service containing an empty request and response.
  - SetBool.srv: Service to set a boolean state to true or false, for enabling or disabling hardware for example.
  - Trigger.srv: Service with an empty request header used for triggering the activation or start of a service.

```
uint32 request
---
uint32 response
```

```
uint32 a
uint32 b
---
uint32 sum
```

Example – Services

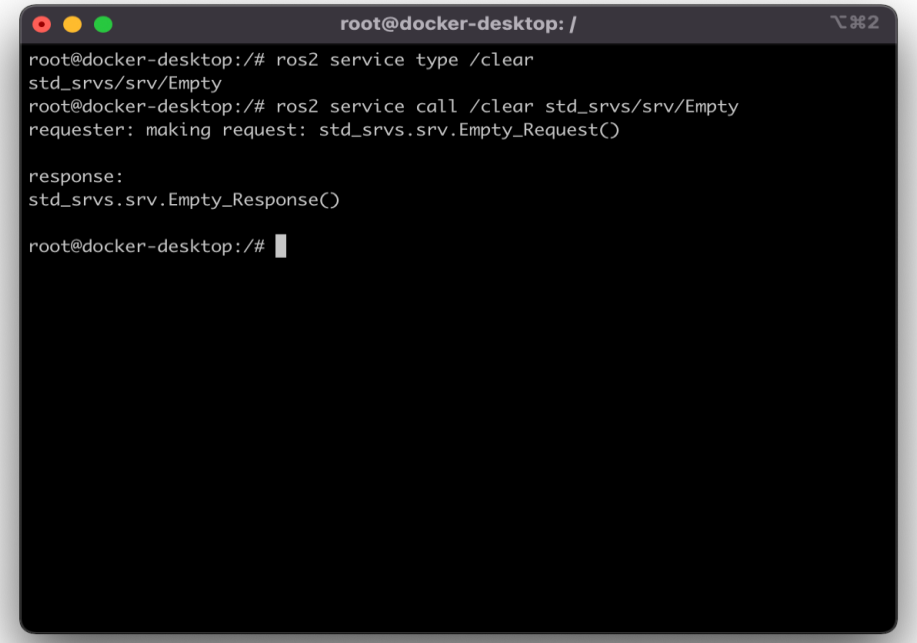
$ ros2 run turtlesim turtlesim_node
$ ros2 service list

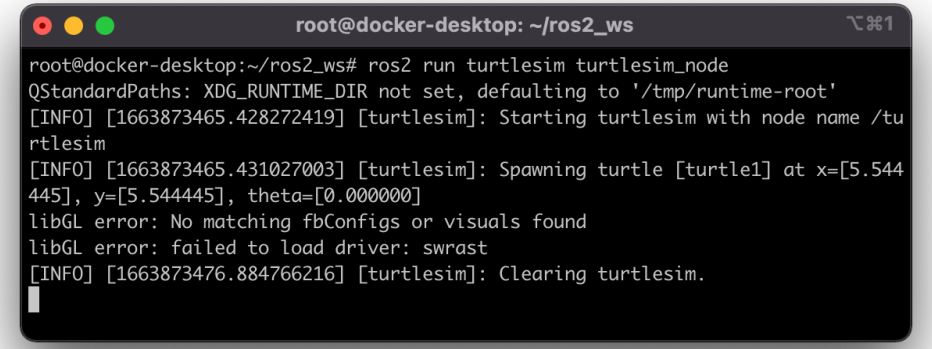


$ ros2 run turtlesim turtle_teleop_key

Example – Services

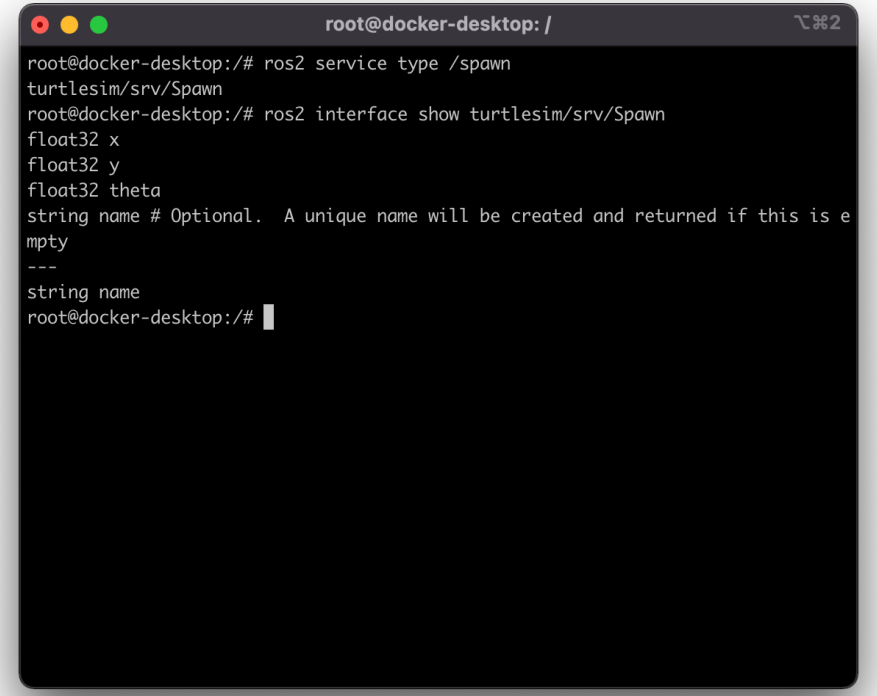
$ ros2 service type /clear
$ ros2 service call /clear std_srvs/srv/Empty





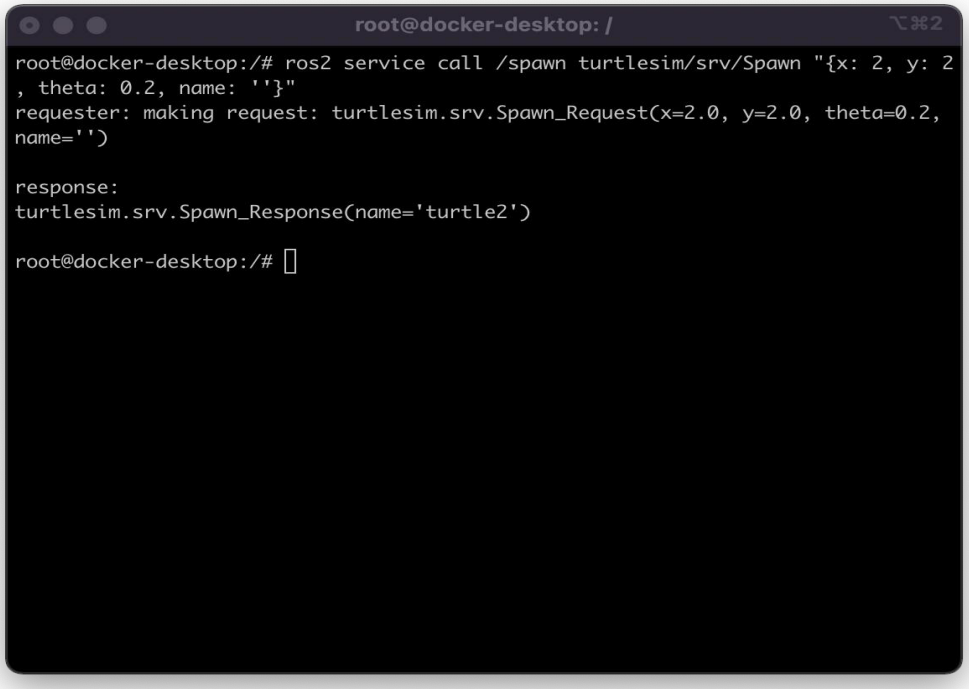
Example – Services

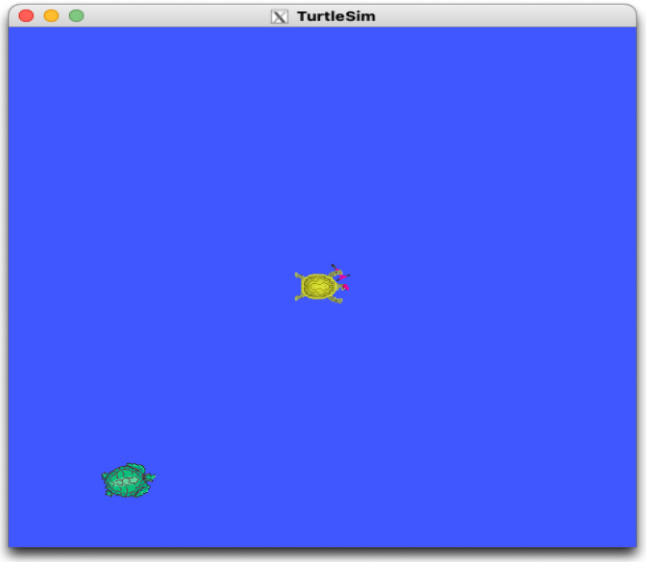
$ ros2 service type /spawn
$ ros2 interface show turtlesim/srv/Spawn



Example – Services

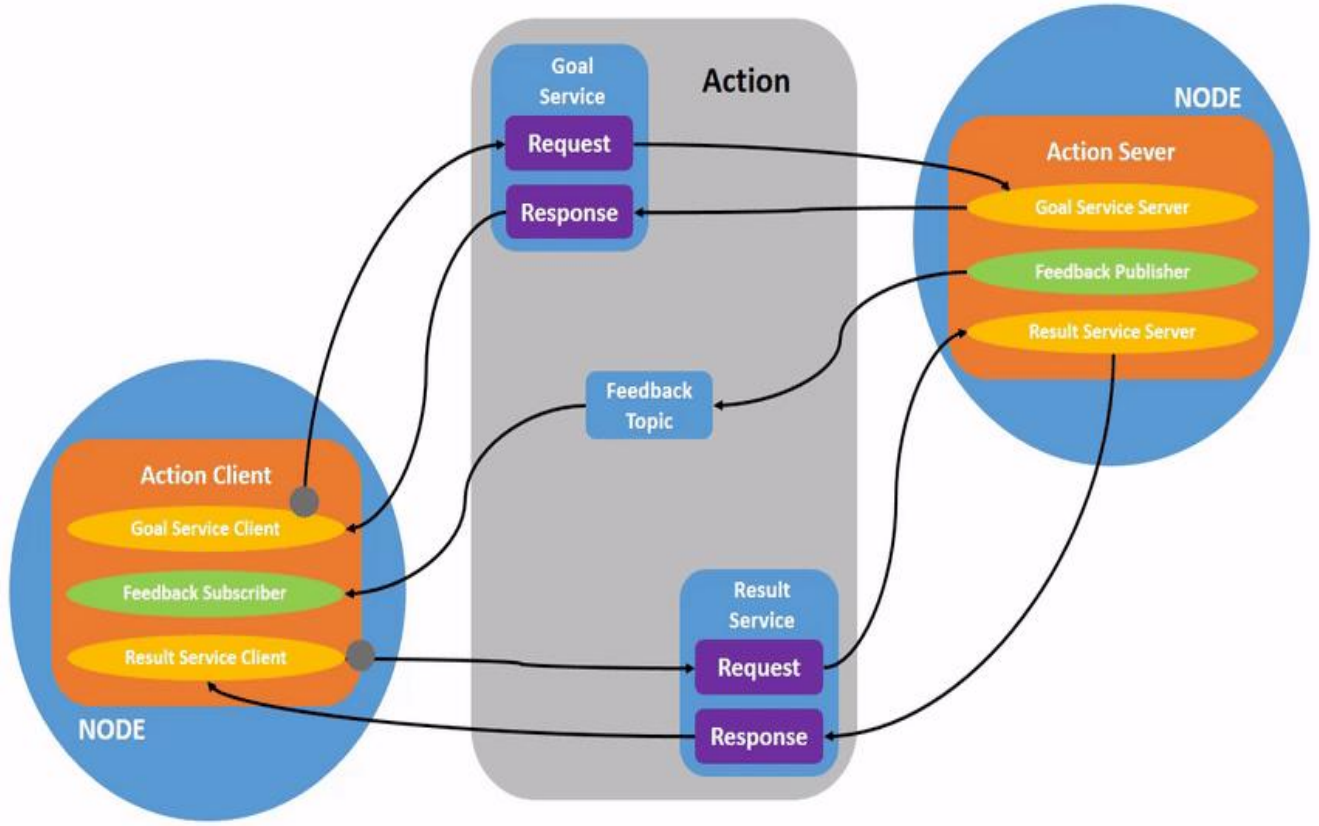
$ ros2 service call /spawn turtlesim/srv/Spawn "{x: 2,y: 2,theta: 0.2,name: 'turtle2'}"





ROS 2 Actions

- Actions consist of three parts: a **goal**, **feedback**, and a **result**
- Actions are built on *topics* and *services*
- An "**action client**" node sends a goal to an "**action server**" node that acknowledges the goal and returns a stream of feedback and a result



Example – Actions

$ ros2 node info /turtlesim

```
elamon@DISI167:~$ ros2 node info /turtlesim
/turtlesim
  Subscribers:
    /parameter_events: rcl_interfaces/msg/ParameterEvent
    /turtle1/cmd_vel: geometry_msgs/msg/Twist
  Publishers:
    /parameter_events: rcl_interfaces/msg/ParameterEvent
    /rosout: rcl_interfaces/msg/Log
    /turtle1/color_sensor: turtlesim/msg/Color
    /turtle1/pose: turtlesim/msg/Pose
  Service Servers:
    /clear: std_srvs/srv/Empty
    /kill: turtlesim/srv/Kill
    /reset: std_srvs/srv/Empty
    /spawn: turtlesim/srv/Spawn
    /turtle1/set_pen: turtlesim/srv/SetPen
    /turtle1/teleport_absolute: turtlesim/srv/TeleportAbsolute
    /turtle1/teleport_relative: turtlesim/srv/TeleportRelative
    /turtlesim/describe_parameters: rcl_interfaces/srv/DescribeParameters
    /turtlesim/get_parameter_types: rcl_interfaces/srv/GetParameterTypes
    /turtlesim/get_parameters: rcl_interfaces/srv/GetParameters
    /turtlesim/list_parameters: rcl_interfaces/srv/ListParameters
    /turtlesim/set_parameters: rcl_interfaces/srv/SetParameters
    /turtlesim/set_parameters_atomically: rcl_interfaces/srv/SetParametersAtomically
  Service Clients:

  Action Servers:
    /turtle1/rotate_absolute: turtlesim/action/RotateAbsolute
```

Example – Actions

$ ros2 run turtlesim turtle_teleop_key
$ ros2 node info /teleop_turtle

```
elamon@DISI167:~$ ros2 node info /teleop_turtle
/teleop_turtle
  Subscribers:
    /parameter_events: rcl_interfaces/msg/ParameterEvent
  Publishers:
    /parameter_events: rcl_interfaces/msg/ParameterEvent
    /rosout: rcl_interfaces/msg/Log
    /turtle1/cmd_vel: geometry_msgs/msg/Twist
  Service Servers:
    /teleop_turtle/describe_parameters: rcl_interfaces/srv/DescribeParameters
    /teleop_turtle/get_parameter_types: rcl_interfaces/srv/GetParameterTypes
    /teleop_turtle/get_parameters: rcl_interfaces/srv/GetParameters
    /teleop_turtle/list_parameters: rcl_interfaces/srv/ListParameters
    /teleop_turtle/set_parameters: rcl_interfaces/srv/SetParameters
    /teleop_turtle/set_parameters_atomically: rcl_interfaces/srv/SetParametersAtomically
  Service Clients:

  Action Servers:

  Action Clients:
    /turtle1/rotate_absolute: turtlesim/action/RotateAbsolute
```

Example – Actions

$ ros2 action list
$ ros2 action list -t
$ ros2 action info /turtle1/rotate_absolute
$ ros2 interface show turtlesim/action/RotateAbsolute
$ ros2 action send_goal <action_name> <action_type> <values>

```
elamon@DISI167:~$ ros2 action list
/turtle1/rotate_absolute
elamon@DISI167:~$ ros2 action list -t
/turtle1/rotate_absolute [turtlesim/action/RotateAbsolute]
elamon@DISI167:~$ ros2 action info /turtle1/rotate_absolute
Action: /turtle1/rotate_absolute
Action clients: 1
    /teleop_turtle
Action servers: 1
    /turtlesim
elamon@DISI167:~$ ros2 interface show turtlesim/action/RotateAbsolute
# The desired heading in radians
float32 theta
---
# The angular displacement in radians to the starting position
float32 delta
---
# The remaining rotation in radians
float32 remaining
```

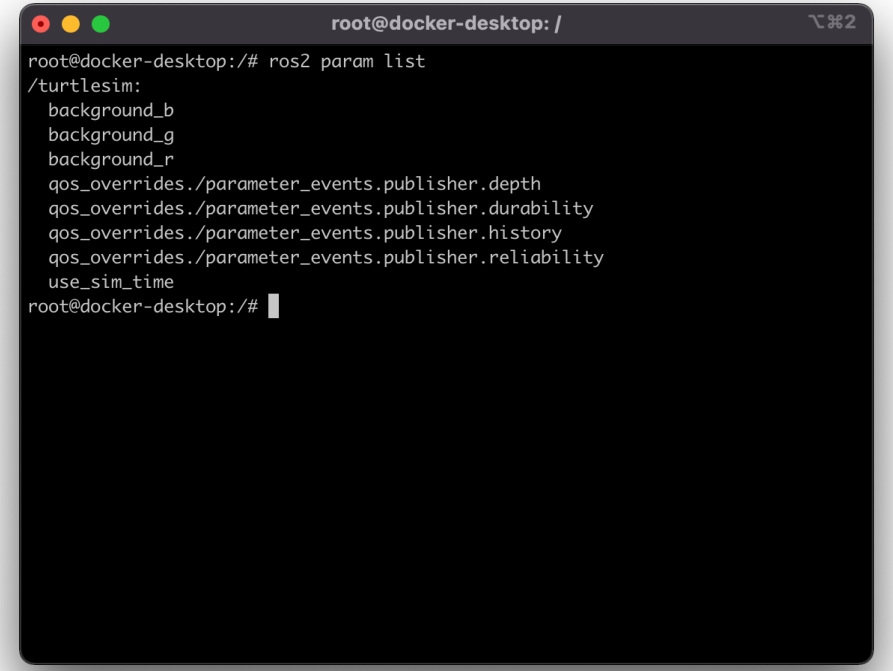
ROS 2 Parameters

- Parameters provide a way to change **settings** of nodes
- ROS2 provides a **parameter server**

$ ros2 param

- Each parameter is associated with a **type**
- They may be organized into **namespaces**

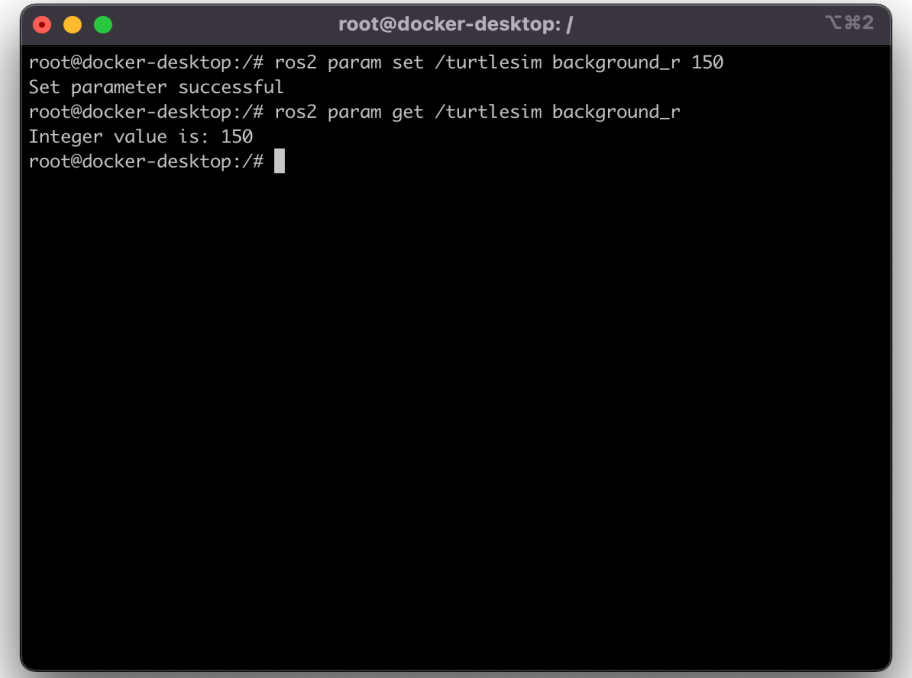
$ ros2 param list

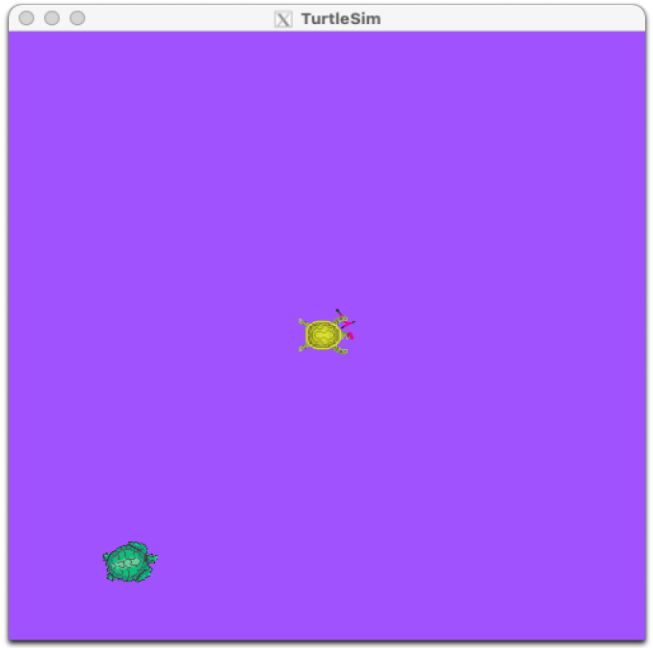


Example – Parameters

$ ros2 param set /turtlesim background_r 150

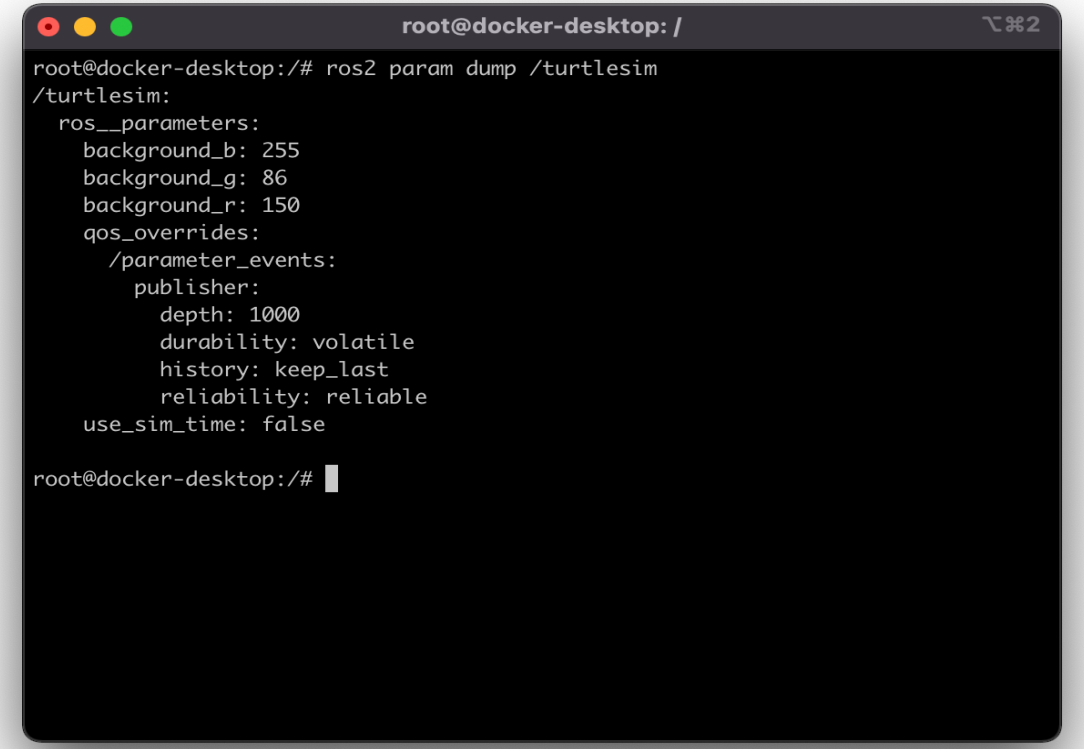
$ ros2 param get /turtlesim background_r





Example – Parameters

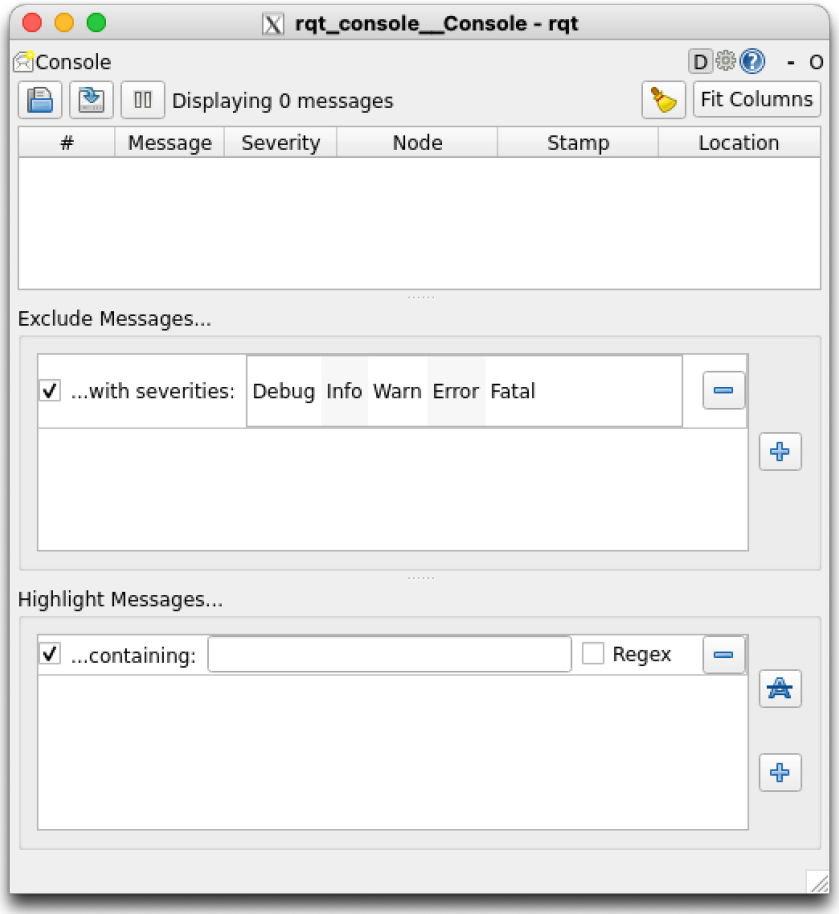
$ ros2 param dump /turtlesim
$ ros2 param dump /turtlesim > turtlesim.yaml
$ ros2 param load /turtlesim turtlesim.yaml



ROS 2 Debugging

rqt_console attaches to the ROS's logging framework to display output from loggers

ros2 run rqt_console rqt_console

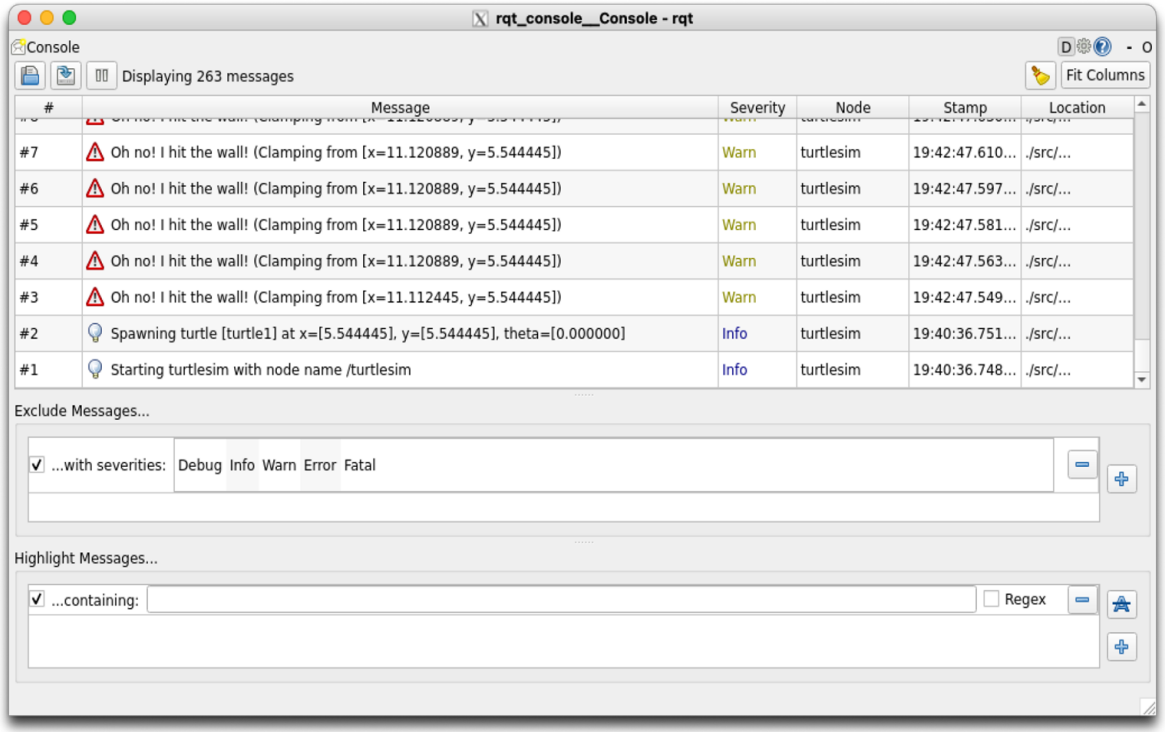


ROS 2 Debugging

Make the turtle move until it crashes into the wall

$ ros2 run turtlesim turtlesim_node

$ ros2 topic pub -r 1 /turtle1/cmd_vel geometry_msgs/msg/Twist
"{linear:{x:2.0,y:0.0,z:0.0}, angular:{x:0.0,y:0.0,z:0.0}}"



When to use each communication interface?

Interface	How it works	When to use it
Topic	One-to-many mechanism. The publishing node specifies a name for the communication channel (the topic) and the type of data (the message) to be transferred.	Whenever your node produces data without requiring any input from the listener and doesn't care if and when anyone will receive it: • Sensor data; • Robot state information; • Diagnostic information,
Service	Client-server mechanism. Multiple clients can send "requests" to a server that will respond to each of them individually.	• If a client expects a server to produce and send back some data that depends on the content of the request. • If a client needs to receive a confirmation that the data was processed correctly by the server. A missed communication between sender and receiver is treated as an error. • If a client needs to verify that the receiver of the data (the server) is ready to process the message before sending it.
Action	Mix of services (sending the goal and receiving the result) and topic (feedback), for long task executions.	Use it whenever you want a different node to execute long tasks asynchronously. In addition, actions allow to: • Cancel a task during its execution. • Provide feedback while executing.

UNIVERSITÀ
DI TRENTO
Department of
Information Engineering and Computer Science

ROS 2 Development

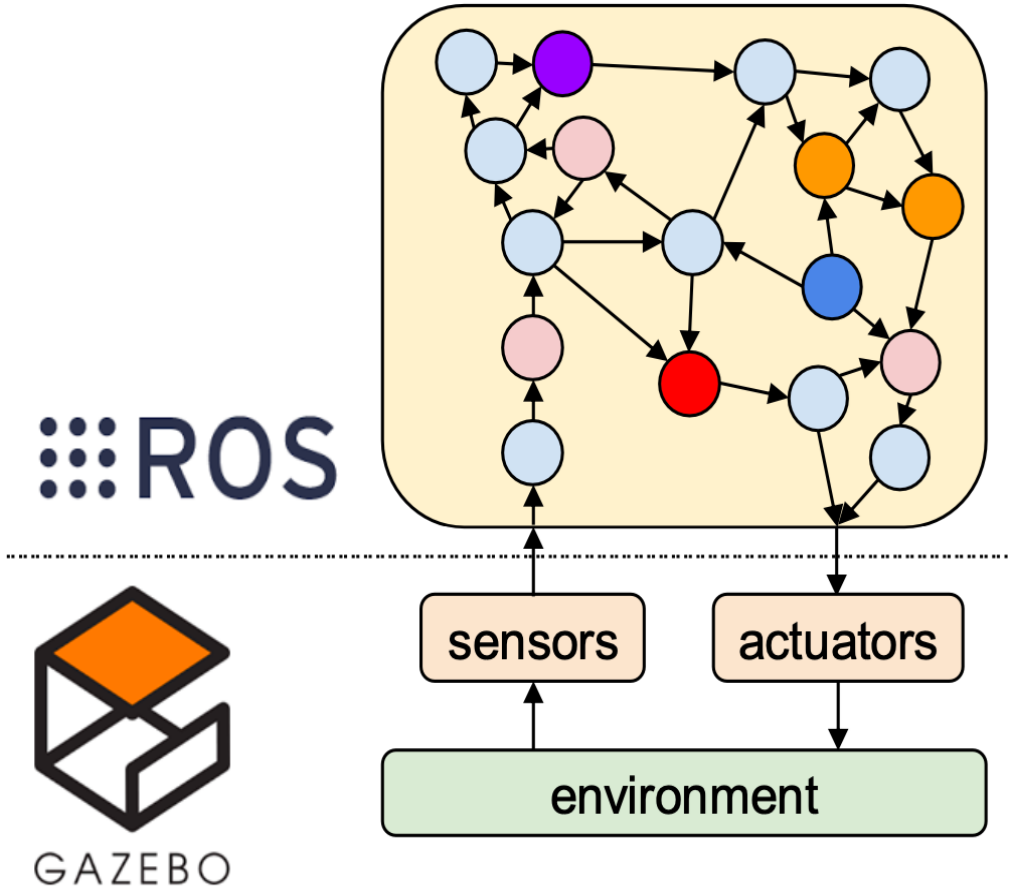
Edoardo Lamon, Luigi Palopoli, Enrico Saccon

Software Development for Collaborative Robots

Academic Year 2025/26

ROS 2 Structure

- A ROS2 project is divided in **workspaces** and **packages**
- A workspace is a collection of other workspaces or packages
- A package is a collection of files and resources related
- A node is an executable that run and communicates with the ROS2 framework
- Packages may contain multiple nodes



Creating the First Workspace

- Prerequisites-install colcon: sudo apt install python3-colcon-common-extensions
- Let's adhere to the standard and create the workspace in ~/ros_ws and add the directory src inside the workspace:
  mkdir -p ~/ros2_ws/src
  cd ~/ros2_ws
- Clone the examples: git clone https://github.com/ros2/examples src/examples -b humble
- Source the ROS 2 libraries (underlay): source /opt/ros/humble/setup.bash
- Run $ colcon build --symlink-install

```
ros2_ws
   ├── build
   ├── install
   ├── log
   └── src
```

Test the compiled examples

- Inside the workspace, source it (overlay):
  source install/setup.bash

- Run a subscriber node from the examples:
  ros2 run examples_rclcpp_minimal_subscriber subscriber_member_function

- In another terminal, source the workspace and run a publisher node:
  ros2 run examples_rclcpp_minimal_publisher publisher_member_function

ROS 2 Nodes

Publishers and Subscribers

Create the First Package

- Create a new package with the following command:

```
$ ros2 pkg create custom_pkg --build-type ament_cmake
```

- --node-name <name> allows to specify also the name of the file containing a node you will create

  e.g. ros2 pkg create custom_pkg --build-type ament_cmake --node-name custom_node

Writing the Publisher Node

ROS1-way

```
int main (int argc, char **argv) {

 rclcpp::init(argc, argv);

 auto node=std::make_shared<rclcpp::Node>(“
  my_publisher”);

 rclcpp::spin(node);
 rclcpp::shutdown();

 return 0;
}
```

ROS2-way

```
class MyPublisher : public rclcpp::Node {
 public:
  MyPublisher() : Node(“my_publisher”) {}
};

int main (int argc, char **argv) {
 rclcpp::init(argc, argv);
 auto node=std::make_shared<MyPublisher>();
 rclcpp::spin(node);
 rclcpp::shutdown();
 return 0;
}
```

Writing the Publisher Node

```
#include "rclcpp/rclcpp.hpp"
#include "std_msgs/msg/string.hpp"

using namespace std::chrono_literals;


class MyPublisher : public rclcpp::Node {
public:
  MyPublisher() : Node(“my_publisher”) {...}
private:
  void timer_callback(){..}
  rclcpp::TimerBase::SharedPtr timer_;
  rclcpp::Publisher<std_msgs::msg::String>::SharedPtr publisher_;
}
```

Writing the Publisher Node

```
class MyPublisher : public rclcpp::Node {

public:
  MyPublisher() : Node("my_publisher") {...}

private:
  void timer_callback() {
    auto message=std_msgs::msg::String();
    message.data="Hello World!";
    publisher_->publish(message);
  }
  rclcpp::TimerBase::SharedPtr timer_;
  rclcpp::Publisher<std_msgs::msg::String>::SharedPtr publisher_;
}
```

Writing the Publisher Node

```
class MyPublisher : public rclcpp::Node {
public:
  MyPublisher() : Node("my_publisher") {
    publisher_ = this->create_publisher<std_msgs::msg::String>(TOPIC_NAME, 10);
    timer_ = this->create_wall_timer(500ms, std::bind(&MyPublisher::timer_callback, this));
  }
private:
  void timer_callback(){
    auto message=std_msgs::msg::String();
    message.data="Hello World!";
    publisher_->publish(message);
  }
  rclcpp::TimerBase::SharedPtr timer_;
  rclcpp::Publisher<std_msgs::msg::String>::SharedPtr publisher_;
}
```

Naming your topic

There are rules to look out when assigning a name to an interface:

- It must not be empty.
- Must only contain alphanumeric symbols and underscores '_' or the forward slash '/'; the latter two should never be repeated and never be at the end of the name.
- Can start with the prefix "~/", the private namespace substitution character (you don't really need it).
- Can use curly brackets '{}' for name substitutions (e.g., "{node}/chatter").

Some topic names are by convention bounded to specific message types and usages. This is not an enforced rule, but to avoid confusion it's good rule to respect it (e.g., 'initialpose', 'joint_states', 'rosout', 'tf' and 'tf_static', etc.)

Compiling the Publisher Node

- Add dependencies to package.xml

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

- Add dependencies to CMakeLists.txt

```
find_package(rclcpp REQUIRED)
find_package(std_msgs REQUIRED)

add_executable(mytalker src/publisher_node.cpp)
ament_target_dependencies(mytalker rclcpp std_msgs)

install(TARGETS
     mytalker
     DESTINATION lib/${PROJECT_NAME})
```

Compiling the Publisher Node

- It’s good practice to run rosdep in the root of your workspace (ros2_ws) to check for missing dependencies before building:

```
sudo apt install python3-rosdep2
rosdep update
rosdep install -i --from-path src --rosdistro humble -y
```

- In the workspace directory, compile with:

```
colcon build
```

- Source the new files so that the client library can find the new nodes:

```
source install/setup.bash
```

Run the Publisher Node

- Run the publisher node:

  $ ros2 run custom_pkg mytalker

- Check the name:

  ros2 node list

- Listen to the messages sent on the topic:

  $ ros2 topic list

  $ ros2 topic echo /topic

  $ ros2 topic hz /my_topic

How to instantiate a ROS 2 node #1

```cpp
#include "rclcpp/rclcpp.hpp"
int main(int argc, char **argv)
{
    rclcpp::init(argc, argv);
    auto node = std::make_shared<rclcpp::Node>("
    best_node_name");
    rclcpp::spin(node);
    rclcpp::shutdown();
    return 0;
}
```

- The node is created as a **shared pointer** to a rclcpp::Node class instantiated with your favourite name.
- This pointer can now be used to read parameters, create publishers, subscribers, and all the other fancy features.
- This solution can prove to be problematic and **should be avoided**!

How to instantiate a ROS 2 node #2

```cpp
#include "rclcpp/rclcpp.hpp"

class OurClass : public rclcpp::Node {
  public:
    OurClass() : Node("best_node_name") {
      // Some constructor
    }
}

int main(int argc, char **argv)
{
    rclcpp::init(argc, argv);
    auto node = std::make_shared<OurClass>();
    rclcpp::spin(node->get_node_base_interface());
    rclcpp::shutdown();
    return 0;
}
```

- **Separation**: The custom OurClass class, which inherits from the Node class, is easy to isolate it into a library that is independent from the execution of a process:
  - The node class contains the *functionality*.
  - The main only bootstraps ROS and runs the Node.

- **Modularity** and **reusability**:
  - The Node class can be reused in different executables or launch configurations.
  - If you change the node's logic, you don't touch main.cpp.
  - You might spin multiple nodes in one process (using executors).

- **Testing**:
  - You can unit-test the Node class directly (without spinning ROS)

How to instantiate a ROS 2 node #3

```cpp
#include "rclcpp/rclcpp.hpp"

namespace our_namespace
{

class OurClass : public rclcpp::Node {
  public:
    OurClass(const rclcpp::NodeOptions & options) : Node("
    best_node_name", options) {
      // Some constructor
    }
}

}
#include "rclcpp_components/register_node_macro.hpp"
RCLCPP_COMPONENTS_REGISTER_NODE(our_namespace::OurClass)
```

- **Components** are the recommended way to create nodes in ROS 2;

- A component can be:
  1. manually integrated into your program at compile time (in the CMakeList.txt);

```
add_library(our_component SHARED
  src/our_file.cpp)

# For Windows compatibility
target_compile_definitions(our_component
  PRIVATE "COMPOSITION_BUILDING_DLL")

ament_target_dependencies(our_lib
  "rclcpp"
  "rclcpp_components")

rclcpp_components_register_nodes(our_component "
    our_namespace::OurClass")
```

  2. loaded into an **executor** at runtime
     - using command line commands;
     - through a launchfile.

How to instantiate a ROS 2 node #3

By making the process layout a deploy-time decision the user can choose between:

- running multiple nodes in **separate processes** with the benefits of **process/fault isolation** as well as **easier debugging** of individual nodes;
- running multiple nodes in a **single process** with the **lower overhead** and optionally **more efficient communication** (see Intra Process Communication).

Exercise

- Re-write and compile the publisher node using separate files for class header, class implementation and main (my_publisher.hpp, my_publisher.cpp, my_publisher_node.cpp) .

- Re-write and compile the publisher node as component.

Useful Links

- CMake build system:
  https://docs.ros.org/en/humble/How-To-Guides/Ament-CMake-Documentation.html
  https://colcon.readthedocs.io/en/released/index.html
- Code style:
  https://docs.ros.org/en/humble/The-ROS2-Project/Contributing/Code-Style-Language-Versions.html
- Documentation, testing:
  https://docs.ros.org/en/humble/The-ROS2-Project/Contributing/Developer-Guide.html#documentation

Writing The Listener

```
class MySubscriber: public rclcpp::Node {
 public:
   MySubscriber() : Node("my_subscriber") {}
};

int main (int argc, char **argv) {
 rclcpp::init(argc, argv);
 auto node=std::make_shared<MySubscriber>();
 rclcpp::spin(node);
 rclcpp::shutdown();
 return 0;
}
```

Writing The Listener

```
class MySubscriber : public rclcpp::Node
{
  public:
    MySubscriber(): Node("my_subscriber")
    {
      subscription_ = this->create_subscription<std_msgs::msg::String>(
      "my_topic", 10, std::bind(&MySubscriber::topic_callback, this, _1));
    }

  private:
    void topic_callback(const std_msgs::msg::String & msg) const
    {
      RCLCPP_INFO(this->get_logger(), "I heard: '%s'", msg.data.c_str());
    }
    rclcpp::Subscription<std_msgs::msg::String>::SharedPtr subscription_;
};
```

Compiling the Subscriber

- No new dependencies to package.xml and CMakeLists.txt needed (`rclcpp, std_msgs`);
- Add the executable and target for the subscriber node in the CMakeLists.txt:

```
add_executable(listener src/subscriber_member_function.cpp)
ament_target_dependencies(listener rclcpp std_msgs)

install(TARGETS
  mytalker
  mylistener
  DESTINATION lib/${PROJECT_NAME})
```

Exercise

- Re-write and compile the subscriber node using separate files for class header, class implementation and main (my_subscriber.hpp, my_subscriber.cpp, my_ subscriber_node.cpp) .

ROS 2 Nodes

Logging messages

Logging in ROS 2

- In a generic C++ program, it is common to use the "standard output stream" with the object std::cout.
- In ROS, a set of utilities were created for the same goal, but with added functionality.
- The **logging subsystem** in ROS 2 aims to deliver logging messages to a variety of targets, including:
  - To the console (if one is attached), for real-time visualisation;
  - To log files on disk (if local storage is available);
  - To the /rosout topic on the ROS 2 network.

Logging in ROS 2

The most common way of printing a message looks like:

```
RCLCPP_INFO ( node - > get_logger () , " Hello ROS ! " ) ;
```

This is composed of three parts:

- A macro (RCLCPP_INFO in this case);
- A logger object;
- The message that you would like to print.

The output is:

```
[ INFO ] [{ timestamp }] [{ logger name }]: " Hello ROS ! "
```

Choosing the Severity Level

Log messages have a **severity level** associated with them: DEBUG, INFO, WARN, ERROR or FATAL, in ascending order.

```
RCLCPP_DEBUG(node->get_logger(), "This is a detailed info for debugging");
RCLCPP_INFO(node->get_logger(), "This is an general information message");
RCLCPP_WARN(node->get_logger(), "This is a warning message");
RCLCPP_ERROR(node->get_logger(), "This is an serious issues message");
RCLCPP_FATAL(node->get_logger(), “The system is unusable message");
```

Choosing the Severity Level

Level	Purpose	Example (Manipulation)	Color
DEBUG	Detailed internal info, useful for debugging specific components. *Disabled by default.*	List of all detected objects on the table.	🟩 Green
INFO	Normal operation info — helps understand robot behavior.	Command received or object being grasped.	⚪ White
WARN	Unexpected events that may reduce performance; system still runs.	Using default parameter because custom value not found.	🟨 Yellow
ERROR	Something failed — robot cannot complete its task, needs recovery.	Object missing → abort task.	🟥 Red
FATAL	Critical failure — robot inoperable, no recovery possible.	Lost connection to motor controller.	🔴 Dark red

Changing the Severity Level

- A logger will only process log messages with severity at or higher than a specified level chosen for the logger.
- The default logging level in ROS 2 is **INFO**;
- You can change the logging level
  - from the command line:
    ros2 run my_package my_node_executable --ros-args --log-level DEBUG
  - in the code:
    node->get_logger().set_level(rclcpp::Logger::Level::Debug);
  - as an environmental variable:
    export RCUTILS_LOGGING_MIN_SEVERITY=DEBUG

Logging Macros in ROS 2

ROS 2 provides different logging options.

Logging style:
- RCLCPP_{DEBUG,INFO,WARN,ERROR,FATAL} - output the given printf-style message every time this line is hit
- RCLCPP_{DEBUG,INFO,WARN,ERROR,FATAL}_STREAM - output the given C++ stream-style message every time this line is hit

When logging:
- RCLCPP_{DEBUG,INFO,WARN,ERROR,FATAL}_ONCE - output only the first time this line is hit
- RCLCPP_{DEBUG,INFO,WARN,ERROR,FATAL}_EXPRESSION - output only if the given expression is true
- RCLCPP_{DEBUG,INFO,WARN,ERROR,FATAL}_FUNCTION - output only if the given function returns true
- RCLCPP_{DEBUG,INFO,WARN,ERROR,FATAL}_SKIPFIRST - output all but the first time this line is hit
- RCLCPP_{DEBUG,INFO,WARN,ERROR,FATAL}_THROTTLE - output no more than the given rate in integer milliseconds

Also combinations of the macros are possible. The full list is available here.

ROS 2 Nodes

Callbacks and Executors

The callback

- It is that part of the code that can be executed only by an executor, or by an invocation of one of the spinning functions (*spin, spin once,* and *spin until future complete*).
- You define what the callbacks do, but you can not strictly enforce the moment in which they will be executed.
- Examples:
  - Subscription callbacks (called whenever a message arrives);
  - Timer callbacks (called at a certain frequency);
  - Service calls, including parameters callbacks;
  - Received client responses.

The executors

- An **executor** is an event loop.
  - It checks all the nodes you've given it for *work to do* (incoming messages, timer events, service requests, etc.)
  - It then runs the corresponding callbacks according to its scheduling rules.
- For example, the code: `rclcpp::spin(node);` is just a shorthand for creating a **SingleThreadedExecutor**, adding the node to it, and spinning.
- When executors are **important**:
  - Single-threaded is safe but may be too slow; multi-threaded gives concurrency but needs careful coding;
  - You can run multiple nodes in the same process by adding them to one executor;
  - In real-time robotics, choosing the right executor affects timing guarantees.

The executors

- **SingleThreadedExecutor:**
  - Default if you just call rclcpp::spin(node).
  - Processes one callback at a time, in a single thread.
  - Simple, avoids race conditions, but may block if a callback is slow.

```
rclcpp::executors::SingleThreadedExecutor exec;
exec.add_node(node1);
exec.add_node(node2);
exec.spin();
```

- **StaticSingleThreadedExecutor** (introduced for deterministic, real-time-ish scenarios)
  - Similar to SingleThreadedExecutor, but builds a static schedule of what to check, avoiding some overhead.

The executors

- **MultiThreadedExecutor**
  - Can run multiple callbacks in parallel, using a thread pool.
  - Useful when callbacks are slow or blocking (e.g. heavy computations, waiting for I/O).
  - Careful: you need thread-safe code (shared variables → mutexes).

```
rclcpp::executors::MultiThreadedExecutor exec(rclcpp::ExecutorOptions(), 4); // 4 threads
exec.add_node(node1);
exec.add_node(node2);
exec.spin();
```

The executors

Example with two nodes.

```cpp
#include "rclcpp/rclcpp.hpp"
#include "talker.hpp"
#include "listener.hpp"

int main(int argc, char * argv[])
{
  rclcpp::init(argc, argv);

  auto talker   = std::make_shared<Talker>();
  auto listener = std::make_shared<Listener>();

  // Option A: single-threaded (one callback at a time)
  // rclcpp::executors::SingleThreadedExecutor exec;

  // Option B: multi-threaded (callbacks may run concurrently)
  rclcpp::executors::MultiThreadedExecutor exec;

  exec.add_node(talker);
  exec.add_node(listener);
  exec.spin();

  rclcpp::shutdown();
  return 0;
}
```

ROS 2 Nodes

Custom Interfaces

Standard and Custom ROS 2 Interfaces

✅ **Use *standard* message types (from common packages) when possible**

- **Easier integration** — works seamlessly with other nodes and teams.
- **Fewer dependencies** — no need to ship custom `.msg` files.
- **Faster development** — no extra compilation or maintenance.

❌ **Avoid misusing standard messages**

Don't force data into an existing type *just because it fits syntactically*. Use a **custom message** if:

- The semantics don't match (e.g., voltages ≠ `geometry_msgs/Quaternion`).
- You can't fill all required fields — receivers can't tell if defaults are meaningful.

Custom ROS 2 Interfaces

To create custom messages/services/actions follow these guidelines to keep your system clean, reusable, and compatible:

1. Use dedicated *interface packages*

- Define message/service/action files **in a separate package**.
- That package should contain **only interface definitions** — no source code.
- Prevents circular dependencies.
- Easier distribution (users only need `.msg` files to communicate).

2. Naming convention

- End interface package names with **`_msgs`** or **`_interfaces`**:
  Example: `my_robot_msgs`, `navigation_interfaces`

3. Represent optional values with vectors

- Clear semantics for “optional” fields.
- Use an **empty vector** to mean “value not provided”:
  Example: float64[<=1] optional_value_name

Custom Messages and Services

- The .msg files are required to be placed in directories called msg inside your package: mkdir msg srv

- In msg (or srv), create a custom definition and fill the data types:
  touch MyMsg.msg (or MySrv.srv)

e.g.
geometry_msgs/Point center
float64 radius

```
int64 a
int64 b
int64 c
---
int64 sum
```

Compile the Custom Message

```
find_package(geometry_msgs REQUIRED)
find_package(rosidl_default_generators REQUIRED)

rosidl_generate_interfaces(${PROJECT_NAME}
  "msg/MyMsg.msg“
  “srv/MySrv.srv"
DEPENDENCIES geometry_msgs # Add packages that above messages depend on, in this case geometry_msgs for My_msg.msg)
ament_export_dependencies(rosidl_default_runtime)
```

Compile the Custom Message

```
# My_msg dependencies
<depend>geometry_msgs</depend>

# Compile-time and Run-time dependencies to generate My_msg
<buildtool_depend>rosidl_default_generators</buildtool_depend>
<exec_depend>rosidl_default_runtime</exec_depend>
<member_of_group>rosidl_interface_packages</member_of_group>
```

Build and Verify

- Compile only a package:
  colcon build --packages-select <pkg_name>

- Souce the overlay to see the new message:
  source install/setup.bash

- Verify that your interface creation worked:
  ros2 interface show <pkg_name>/msg/MyMsg
  ros2 interface show <pkg_name>/srv/MySrv

Using Custom Msgs/Srvs in your Package

- If the msg is defined in an **external package** (e.g. custom_msgs), to link it to your node:
  - In CMakeLists.txt we need to add it as a dependency:

```
find_package(custom_msgs REQUIRED)


add_executable(my_publisher_node src/my_publisher_node.cpp)
ament_target_dependencies(my_publisher_node custom_msgs <other dependencies>)
```

  - Also package.xml should be informed

```
<depend>custom_msgs</depend>
```

  - Include the headers which have the form

```
#include "<package_name>/msg/<interface_file_name>.hpp"
#include "custom_msgs/msg/my_msg.hpp"
```

Using Custom Msgs/Srvs in your Package

- If the msg is defined in the **same package** (e.g. custom_pkg), to link it to your node:
  - In CMakeLists.txt:

```
add_executable(my_publisher_node src/my_publisher_node.cpp)
rosidl_get_typesupport_target(cpp_typesupport_target ${PROJECT_NAME}
rosidl_typesupport_cpp)
target_link_libraries(my_publisher_node "${cpp_typesupport_target}")
ament_target_dependencies(my_publisher_node <other dependencies>)
```

  - No additions to the package.xml are required because the generated files are in the same pkg!
  - Include the headers which have the form

```
#include "<package_name>/msg/<interface_file_name>.hpp"
#include "custom_pkg/msg/my_msg.hpp"
```

Exercise

- Use the new custom message in the Publisher node by adding a new Publisher broadcasting on a new topic with message type MyMsg.msg.

ROS 2 Nodes

Parameters

Setting the Parameters

```
class MyPublisher : public rclcpp::Node {
public:
  MyPublisher() : Node("mypublisher"){
    publisher_ = this->create_publisher<std_msgs::msg::String>(TOPIC_NAME, 10);
    timer_ = this->create_wall_timer(500ms, std::bind(&MyPublisher::timer_callback, this));
  }
private:
  void timer_callback(){
    auto message=std_msgs::msg::String();
    message.data="Hello World!";
    publisher_->publish(message);
  }
  rclcpp::TimerBase::SharedPtr timer_;
  rclcpp::Publisher<std_msgs::msg::String>::SharedPtr publisher_;
}
```

Setting the Parameters

```
class MyPublisher : public rclcpp::Node {
public:
  MyPublisher() : Node(“mypublisher”), topic_("default_topic") {
    this->declare_parameter("topic", topic_);
    publisher_ = this->create_publisher<std_msgs::msg::String>(topic_, 10);
    timer_ = this->create_wall_timer(500ms, std::bind(&MyPublisher::timer_callback, this));
  }
private:
  void timer_callback(){
    auto message=std_msgs::msg::String();
    message.data=“Hello World!”;
    publisher_->publish(message);
  }
  std::string topic_;
  rclcpp::TimerBase::SharedPtr timer_;
  rclcpp::Publisher<std_msgs::msg::String>::SharedPtr publisher_;
}
```

Setting the Parameters

```
void MyPublisher::timer_callback(){
  auto message=std_msgs::msg::String();
  message.data="Hello World!";
  publisher_->publish(message);

  std::string newTopic = this->get_parameter("topic").as_string();
  if (newTopic != this->topic_) {
    this->topic_ = newTopic;
    this->publisher_.reset();
    this->publisher_ = this->create_publisher<std_msgs::msg::String>(this->topic_, 10);
  }
}
```

Setting the Parameters

- Get the parameter:

  $ ros2 param get /mypublisher topic

- Listen to verify that the data is arriving correctly:

  $ ros2 topic echo default_topic

- Change the parameter:

  $ ros2 param set /mypublisher topic important_topic

Why Parameters?

- Parameters are "*settings*" for the node
- Allow to dynamically change the configuration of a node
  - Used in launch files
  - Depend on the **context** in which the node is executed
- Sometimes you don't know the actual value of a variable at *compile time*
- They are **associated** to the node
  - ROS1 used the Parameter Server
  - ROS2 access to a local dictionary, eliminating the overhead of network access *

* it's possible to use topics and service to subscribe to other nodes' parameters and monitor their changes, but should be avoided.

How to Access Parameters

- Declare a parameter:

```
auto value = this->declare_parameter("topic_name", "default_value");
```

- Each parameter can be declared **only once**
- Initialization of parameter:
  - default_value
  - Override (declare_parameter function returns the value of the parameter updated with the overrides):
    - Launch file
    - YAML file (saved with dump command in *config* directory):

      ros2 run <pkg> <node> --ros-args --params-file <path to YAML file>

    - Arguments provided through command line:

      ros2 run <pkg> <node> --ros-args -p <param_name>:=<param_value>

How to Access Parameters

- Declare a parameter:

```
auto value = this->declare_parameter("topic_name", "default_value");
```

- Retrieve the parameter value:

```
auto value = this->get_parameter("topic _name ");
```

- Change the parameter:

```
this->set_parameter("topic _name", "new_value");
```

Parameter Types and Description

- Parameters have types:
  - double
  - integer
  - string
  - list → std::vector
- Can have a description:
  - read_only: if true, the parameter value **cannot** be changed at run-time
  - dynamic_typing: if true, it's possible to **change** the parameter type
  - FloatingPointRange/IntegerRange: limit the **range** of the parameter's value
  - additional_constraints: additional constraints

```
auto param_desc = rcl_interfaces::msg::ParameterDescriptor{};
param_desc.description = "This parameter is mine!";

this->declare_parameter("my_parameter", "world", param_desc);
```

Parameter Types and Description

- Parameters have types:
  - double
  - integer
  - string
  - list → std::vector
- Can have a description:
  - read_only: if true, the parameter value **cannot** be changed at run-time
  - dynamic_typing: if true, it's possible to **change** the parameter type
  - FloatingPointRange/IntegerRange: limit the **range** of the parameter's value
  - additional_constraints: e.g. your parameter is a string that can assume only specific values, these could be listed here in a comma separated list.

If your parameter controls important functions and will be updated at run-time, then remember to set the range!

```
auto param_desc = rcl_interfaces::msg::ParameterDescriptor{};
param_desc.description = "This parameter is mine!";

this->declare_parameter("my_parameter", "world", param_desc);
```

Handling Default Parameters in ROS 2

PROBLEM: If you always assign *default values* to parameters, you can't tell whether a parameter was **overridden by the user** or is still using the **default**.

EXAMPLE: Typos or wrong namespaces in YAML files cause silent failures. The node starts, but behaves incorrectly.

SOLUTIONS:

Approach	Behavior	Pros / Cons
1️⃣ **No defaults**	No override → exception at runtime.	✅ Forces correct parameter definition. ❌ Node may crash if parameters missing.
2️⃣ **Warn on missing override**	Use default but log a **warning**.	✅ Prevents crashes. ✅ Makes issues visible. 💡 Can be done by extending `declare_parameter()`.

Dynamic Parameters Updates

- Dynamic parameters updates
  - Repeatedly call `get_parameter()`
  - Subscribe to parameter changes
    - Self
    - Other's

- Having too many parameters might create confusion, so don't abuse them!
  - Use namespaces or parameter groups to group them;
  - Try to generate mechanisms for **automatic parameter tuning.**

Dynamic Parameter Updates

Use ParameterEventHandler → subscribes to parameter changes:

```
// Create a parameter subscriber to detect changes in the parameters
this->param_subscriber_ = std::make_shared<rclcpp::ParameterEventHandler>(this);

// Define the callback to be called when the parameter changes
auto cb = [this](const rclcpp::Parameter & p){
  this->timer_.reset();
  this->timer_ = this->create_wall_timer(std::chrono::milliseconds(p.as_int()), std::bind(&MyPublisher::timer_callback, this));
};
// Specify to which parameter changes to subscribe and which callback to call
this->param_cb_handle_ = this->param_subscriber_->add_parameter_callback("rate", cb);
...
private:
  std::shared_ptr<rclcpp::ParameterEventHandler> param_subscriber_;
  std::shared_ptr<rclcpp::ParameterCallbackHandle> param_cb_handle_;
```

Parameters Without Declaration

- ROS1 allowed for parameters **without declaration**
- Not mandatory in ROS2, but *strongly discouraged* (aka, declare them)
  - Static typing
  - Additional constraints
  - Control over multiple declarations
  - Less prone to errors

UNIVERSITÀ
DI TRENTO
Department of
Information Engineering and Computer Science

ROS 2 Development

Edoardo Lamon, Luigi Palopoli, Enrico Saccon

Software Development for Collaborative Robots

Academic Year 2025/26

ROS 2 Nodes

Launch files

Launch Files

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

- Launch files written in *Python*, *XML*, or *YAML* can start and stop different nodes as well as trigger and act on various events.

- As standard, launch files are located in the *launch* folder inside the package.

Write a Launch File

PYTHON

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

def generate_launch_description():
  return LaunchDescription([
    Node(
      package="chatters",
      executable="mytalker",
      name="mytalker_launch",
      parameters=[
        {"rate"  : 5000}
      ],
      output="screen",
      emulate_tty=True
      remappings=[
       ('/topic', '/topic_launch'),
    )
  ])
```

Write a Launch File

PYTHON

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

def generate_launch_description():
  return LaunchDescription([
    Node(
      package="chatters",
      executable="mytalker",
      name="mytalker_launch",
      parameters=[
        {"rate"  : 5000}
      ],
      output="screen",
      emulate_tty=True
      remappings=[
       ('/topic', '/topic_launch'),
    )
  ])
```

YAML

```yaml
launch:
 - node:
   pkg: "chatters"
   exec: "mytalker"
   name: "mytalker_yaml"
   output: "screen"
   remap:
    -
      from: "/my_topic"
      to: "/default_topic"
param:
  -
    name: "rate"
    value: 5000
```

Write a Launch File

PYTHON

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

def generate_launch_description():
  return LaunchDescription([
    Node(
      package="chatters",
      executable="mytalker",
      name="mytalker_launch",
      parameters=[
        {"rate"  : 5000}
      ],
      output="screen",
      emulate_tty=True
      remappings=[
       ('/topic', '/topic_launch'),
    )
  ])
```

YAML

```yaml
launch:
 - node:
  pkg: "chatters"
  exec: "mytalker"
  name: "mytalker_yaml"
  output: "screen"
  remap:
   -
     from: "/my_topic"
     to: "/default_topic"
param:
  -
   name: "rate"
   value: 5000
```

XML (ROS 1-like)

```xml
<launch>

 <arg name="rate" default="5000"/>

 <node pkg="chatters" exec="mytalker"
name="mytalker_xml" output='screen'>
  <param name="rate " value="$(var
rate)"/>
  <remap from="/topic" to="$(var topic)"/>
 </node>

</launch>
```

Python, XML, or YAML: Which should I use?

- Launch files in ROS 1 were written in XML, so XML may be the most familiar to people coming from ROS 1.
- For most applications, the choice of which ROS 2 launch format comes down to developer preference.
- Using Python for ROS 2 launch is more flexible because of following two reasons:
  - Python is a scripting language, and thus you can leverage the language and its libraries in your launch files;
  - ros2/launch (general launch features) and ros2/launch_ros (ROS 2 specific launch features) are written in Python and thus you have lower level access to launch features that may not be exposed by XML and YAML.

Running the Launch File

- Directly in the *launch* folder (works without installation):

```
cd launch
ros2 launch <launch_file>
```

- Provided by a package (requires installation - RECCOMENDED):

```
ros2 launch <package_name> <launch_file>
```

In the CMakeLists.txt add:

```
install(
    DIRECTORY launch
    DESTINATION share/${PROJECT_NAME}
  )
```

- In ROS 2 the launch file goes with his extension (.py/.xml/.yaml). To ensure ensures that all launch file formats are recognized, add in package.xml:

```
<exec_depend>ros2launch</exec_depend>
```

Exercise

Add the subscriber to the nodes to be launched in each launch file (Python, YAML, XML).

Launch Different Nodes - Python

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

def generate_launch_description():
  talker_node = Node(
    package="chatters",
    executable="mytalker",
    name="mytalker_launch",
    output="screen",
    emulate_tty=True,
    parameters=[
      {"topic" : "topic_launch"},
      {"rate"  : 500}
    ]
  )

listener_node = Node(
    package="chatters",
    executable="mylistener",
    name="mylistener_launch",
    output="screen",
    emulate_tty=True,
    parameters=[
      {"topic" : "topic_launch"}
    ]
)

return LaunchDescription([
    talker_node,
    listener_node
])
```

Launch Different Nodes - YAML

```
launch:
 - node:
   pkg: "chatters"
   exec: "mytalker"
   name: "mytalker_yaml"
   output: "screen"
   param:
   -
     name: "topic"
     value: "topic_yaml"
   -
     name: "rate"
     value: 5000
 - node:
   pkg: "chatters"
   exec: "mylistener"
   name: "mylistener_yaml"
   output: "screen"
   param:
   -
     name: "topic"
     value: "topic_yaml"
```

Launch Different Nodes - XML

```
<launch>
  <arg name="topic_name" default="topic_launch_xml"/>

  <node pkg="custom_pkg" exec="custom_pub_standalone" name="my_publisher_launch_xml"
output='screen'>
    <param name="topic_name" value="$(var topic_name)"/>
  </node>

  <node pkg="custom_pkg" exec="custom_sub_standalone" name="my_subscriber_launch_xml"
output='screen
      <remap from="/my_topic" to="$(var topic_name)"/>
  </node>


</launch>
```

Setting Arguments

From command line it is possible to pass arguments to the launch file with key:=value syntax:

ros2 launch <package_name> <launch_file_name> key:=value

You need to explicitly declare them. In Python:

```
from launch.actions import DeclareLaunchArgument
from launch.substitutions import LaunchConfiguration, TextSubstitution
def generate_launch_description():
    topic_name_arg = DeclareLaunchArgument("topic_name_arg", default_value=TextSubstitution(text="topic_launch"))
    pub_node = Node(
        package='custom_pkg',
        executable='custom_pub_standalone',
        name='my_publisher_launch',
                    parameters=[{'topic_name' : LaunchConfiguration('topic_name_arg')}] )
    return LaunchDescription([
        topic_name_arg,
        pub_node,])
```

Load Parameters File

- We know it is possible to pass to a node a YAML file which contains parameters:

  ros2 run <pkg> <node> --ros-args --params-file <path to YAML file>

- Can we pass it also to a launch file?

Load Parameters File

Can we pass it also to a launch file? YES

```
import os
from ament_index_python import get_package_share_directory

parameters_file = os.path.join(get_package_share_directory('custom_pkg'),
'config','my_publisher.yaml’)

pub_node = Node(
    package='custom_pkg',
    executable='custom_pub_standalone',
    name='my_publisher_launch',
    parameters=[parameters_file]
)
```

Load Parameters File

Can we pass it also to a launch file? YES

CAVEATS:

- Make sure the YAML file is installed, otherwise the it won't be found. In the CMakeLists.txt:

```
install(
DIRECTORY config
DESTINATION share/${PROJECT_NAME})
```

- Make sure the name of the node (e.g. 'my_publisher_launch'), matches the one in the YAML file.

Include Another Launch

It is also possible to nest the call of another launch file:

```
from launch.launch_description_sources import PythonLaunchDescriptionSource, AnyLaunchDescriptionSource


# include another launch file
    launch_include_py = IncludeLaunchDescription(
        PythonLaunchDescriptionSource(os.path.join(
            get_package_share_directory('custom_pkg'),'launch','publisher_launch_python.py'))
    )
    launch_include_yaml = IncludeLaunchDescription(
        AnyLaunchDescriptionSource(os.path.join(
            get_package_share_directory('custom_pkg'),'launch','publisher_launch_yaml.yaml'))
    )
    return LaunchDescription([
        launch_include_py,
        launch_include_yaml
    ])
```

ROS 2 Nodes

Services: clients and servers

Services

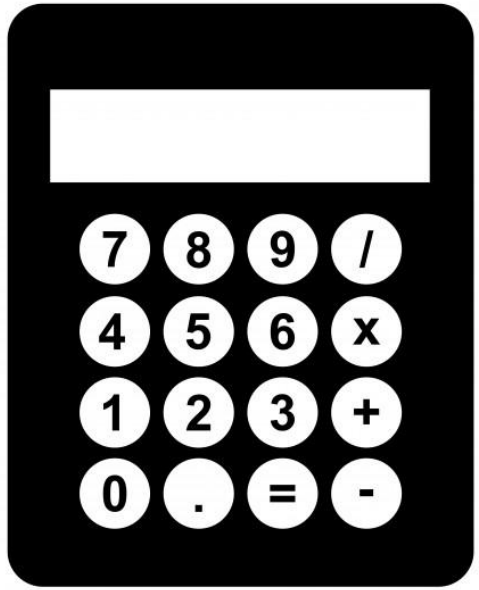
- Actions to be carried out in a **small amount** of time
- In ROS1 services were **synchronous** (client blocking call)
  - If a server did not return a response, the caller would stay hanging ;
  - The server must return an answer almost immediately.
- In ROS2 services are **asynchronous**
  - The response is returned through a future object;
  - The client can continue its execution and check for the result whenever it prefers.
- Best practice: in the client, wait for the response but **set a timeout**

Services

- **Use** if:
  - The data that the server sends to the client *depend* on the request
  - The client needs to receive confirmation that the server processed the data correctly, otherwise an error is returned
  - The client needs to verify that the server is ready to process the message before sending it
- **Not use** if:
  - The server needs a lot of time to process the data → depends on your system
  - The client needs to receive information from the server during its execution

Writing the Service Server

- Let’s build a node that acts as a calculator!
- We want it to be able to do:
  - Additions
  - Subtractions
  - Multiplications
  - Divisions!



Writing the Service Server

```
#include "rclcpp/rclcpp.hpp"
#include <memory>

class Calculator : public rclcpp::Node {
public:
 Calculator() : Node("calculator") {
  this->service_ = this->create_service<>(
    "calculator", std::bind(&Calculator::calc, this,
    std::placeholders::_1, std::placeholders::_2))
 }
private:
 void calc(const std::shared_ptr<::Request> request,
         std::shared_ptr<::Response> response){}
 rclcpp::Service<>::SharedPtr service_;
}
```

```
int main(int argc, char **argv)
{
 rclcpp::init(argc, argv);
 auto node =
std::make_shared<Calculator>();
 rclcpp::spin(node);
 rclcpp::shutdown();
}
```

Writing the Service Server

```
#include "rclcpp/rclcpp.hpp"
#include <memory>

class Calculator : public rclcpp::Node {
public:
 Calculator() : Node("calculator") {
   this->service_ = this->create_service<>(
     "calculator", std::bind(&Calculator::calc, this,
     std::placeholders::_1, std::placeholders::_2))
 }
private:
 void calc(const std::shared_ptr<::Request> request,
          std::shared_ptr<::Response> response){}
 rclcpp::Service<>::SharedPtr service_;
}
```

What should we put here?

Writing the Service Server

```
#include "rclcpp/rclcpp.hpp"
#include <memory>

class Calculator : public rclcpp::Node {
public:
 Calculator() : Node("calculator") {
   this->service_ = this->create_service<>(
     "calculator", std::bind(&Calculator::calc, this,
     std::placeholders::_1, std::placeholders::_2))
 }
private:
 void calc(const std::shared_ptr<::Request> request,
          std::shared_ptr<::Response> response){}
 rclcpp::Service<>::SharedPtr service_;
}
```

```
string type
float64 a
float64 b
---
float64 res
```

Service Interface

- Where should it be placed? <package_name>/srv/Calc.srv
  - <package_name> → other package → BEST PRACTICE
  - <package_name> → same package

- How should it be added for compilation?

  Package.xml

```
<buildtool_depend>rosidl_default_generators</buildtool_depend>
<exec_depend>rosidl_default_runtime</exec_depend>
<member_of_group>rosidl_interface_packages</member_of_group>
```

  CMakeLists.txt

```
find_package(rosidl_default_generators REQUIRED)
rosidl_generate_interfaces(${PROJECT_NAME} "srv/Calc.srv")
```

- Compile with colcon build and check that it is correct:

  $ source install/setup.bash && ros2 interface show <pkg_name>/srv/Calc

package/srv/Calc.srv

```
string type
float64 a
float64 b
---
float64 res
```

Service Interface

- Create a new package and a folder srv inside
- How should it be included in the source files?
- In CMakeLists.txt we need to add it as a dependency:

```
find_package(calculator_interfaces REQUIRED)
ament_target_dependencies(calculator_node rclcpp calculator_interfaces)
```

- If the package **is not** the same, then also package.xml should be informed

```
<depend>calculator_interfaces</depend>
```

- Include the headers which have the form

```
#include "<package_name>/srv/<interface_file_name>.hpp"
#include "calculator_interfaces/srv/calc.hpp"
```

Write the Service Server

```
#include "rclcpp/rclcpp.hpp"
#include <memory>
#include "calculator_interfaces/srv/calc.hpp"

class Calculator : public rclcpp::Node {
public:
 Calculator() : Node("calculator") {
  this->service_ = this->create_service<calculator_interfaces::srv::Calc>(
    "calculator", std::bind(&Calculator::calc, this,
    std::placeholders::_1, std::placeholders::_2));
 }
private:
 void calc(const std::shared_ptr<calculator_interfaces::srv::Calc::Request> request,
         std::shared_ptr<calculator_interfaces::srv::Calc::Response> response){}
 rclcpp::Service<calculator_interfaces::srv::Calc>::SharedPtr service_;
};
```

Write the Service Server

```
void calc(const std::shared_ptr<calculator_interfaces::srv::Calc::Request> request,
        std::shared_ptr<calculator_interfaces::srv::Calc::Response> response)
{
  if (request->type == "sum"){
    response->res = request->a + request->b;
  }
  else if (request->type == "sub"){
    response->res = request->a - request->b;
  }
  else if (request->type == "mul"){
    response->res = request->a * request->b;
  }
  else if (request->type == "div"){
    response->res = request->a / request->b;
  }
  else {
    RCLCPP_ERROR(this->get_logger(), "Invalid operation type");
    return;
  }
  RCLCPP_INFO(this->get_logger(), "Operation: %s %s %s = %s",
    std::to_string(request->a).c_str(), request->type.c_str(),
    std::to_string(request->b).c_str(), std::to_string(response->res).c_str());
}
```

Write the Service Client

```cpp
#include "rclcpp/rclcpp.hpp"
#include "calculator_interfaces/srv/calc_msg.hpp“

using namespace std::chrono_literals;
using namespace calculator_interfaces::srv;
using namespace std::chrono;

const auto TIMEOUT = 1s;

int main(int argc, char **argv)
{
 rclcpp::init(argc, argv);

 if (argc != 4) {
    RCLCPP_INFO(rclcpp::get_logger("calculator_client"), "usage: calculator_client X T Y");
    return 1;
 }

 std::shared_ptr<rclcpp::Node> node = rclcpp::Node::make_shared("calculator_client");
 rclcpp::Client<Calc>::SharedPtr client =
  node->create_client<Calc>("calculator");

 auto request = std::make_shared<Calc::Request>();
 request->a = atoll(argv[1]);
 request->b = atoll(argv[3]);

 switch (argv[2][0]){
 case '+':
  request->type = "sum"; break;
 case '-':
  request->type = "sub"; break;
 case '*':
  request->type = "mul"; break;
 case '/':
  request->type = "div"; break;
 default:
  RCLCPP_ERROR(rclcpp::get_logger("calculator_client"), "Invalid operation type");
  break;
 }
 while (!client->wait_for_service(TIMEOUT)) {
  if (!rclcpp::ok()) {
   RCLCPP_ERROR(rclcpp::get_logger("calculator_client"), "Interrupted while waiting for the service. Exiting.");
   return 0; }
  RCLCPP_INFO(rclcpp::get_logger("calculator_client"), "service not available, waiting again...");
 }
```

Write the Service Client

```
auto result = client->async_send_request(request);

if (rclcpp::spin_until_future_complete(node, result) ==
  rclcpp::FutureReturnCode::SUCCESS)
{
  RCLCPP_INFO(rclcpp::get_logger("calculator_client"), "Result of %lf %s %lf = %lf",
    request->a, request->type.c_str(), request->b, result.get()->res);
} else {
  RCLCPP_ERROR(rclcpp::get_logger("calculator_client"), "Failed to call service  calculator");}
 rclcpp::shutdown();
return 0;
}
```

- Use always the timeout for receiving the response (in some examples it is not present)!

- To invoke the callbacks of subscriptions, timers, service servers, action servers, etc. on incoming messages and events on one or multiple threads, use an **Executor.**

Write the Service Client

- Be aware : spin_until_future_complete() can not be used if the node is added as a component to an **Executor;**

- In that case, one might store the result and check it from a timer with result.wait_for(DELAY) (non-blocking):

```
auto result = client->async_send_request(request);
auto start_time = get_clock()->now();
using namespace std::chrono_literals;
const auto timeout = 10s;
while(result.wait_for(1s) != rclcpp::FutureReturnCode::
    SUCCESS && rclcpp::ok())
{
    // Any other code to execute while waiting
    if(get_clock()->now() - start_time > timeout) {
      RCLCPP_ERROR(get_logger(), "Timeout elapsed waiting
    for service");
      break;
    }
}
if(get_clock()->now() - start_time < timeout)
    auto my_result = result.get();
else
    // handle failure
```

Exercise

- Create a launch file where both nodes are launched and the parameters are set as arguments in the launch file.

UNIVERSITÀ
DI TRENTO
Department of
Information Engineering and Computer Science

ROS 2 Development

Edoardo Lamon

Software Development for Collaborative Robots

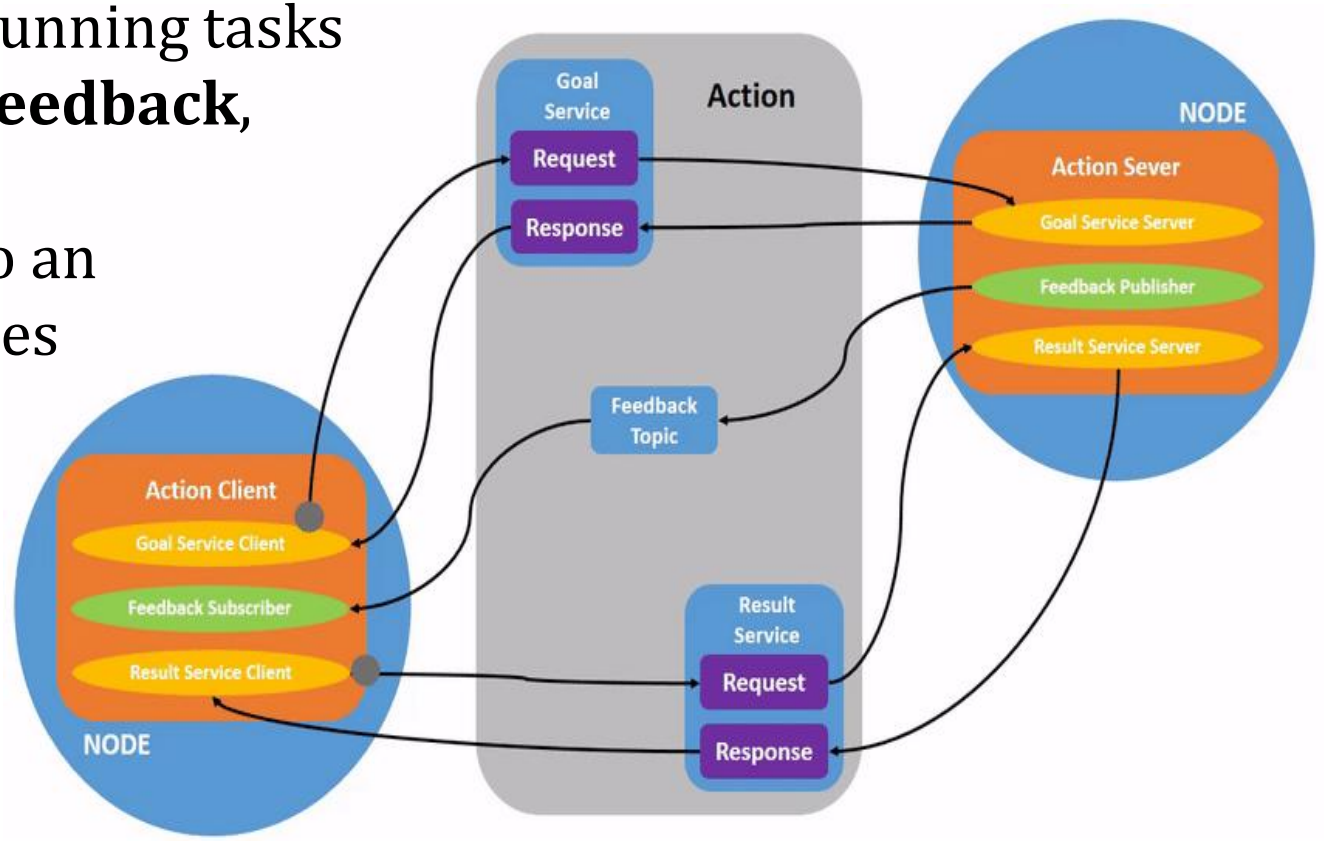
Academic Year 2025/26

ROS 2 Nodes

Actions: clients and servers

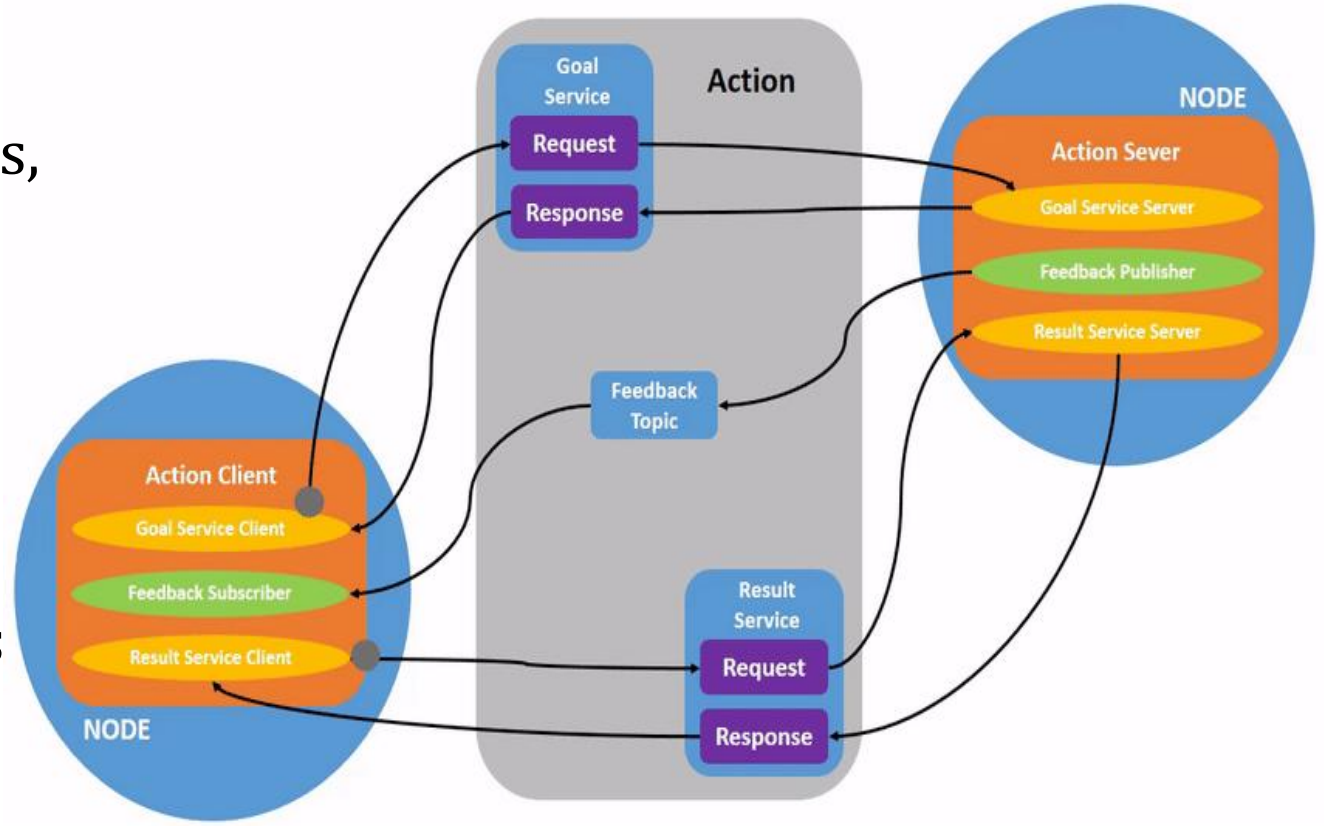
ROS 2 Actions

- ROS (2) actions are intended for long running tasks
- Actions consist of three parts: a **goal**, **feedback**, and a **result**
- An "**action client**" node sends a goal to an "**action server**" node that acknowledges the goal and returns a stream of feedback and a result
- Each goal runs **asynchronously** (server spawns a thread)
- Threads don't block the **executor**



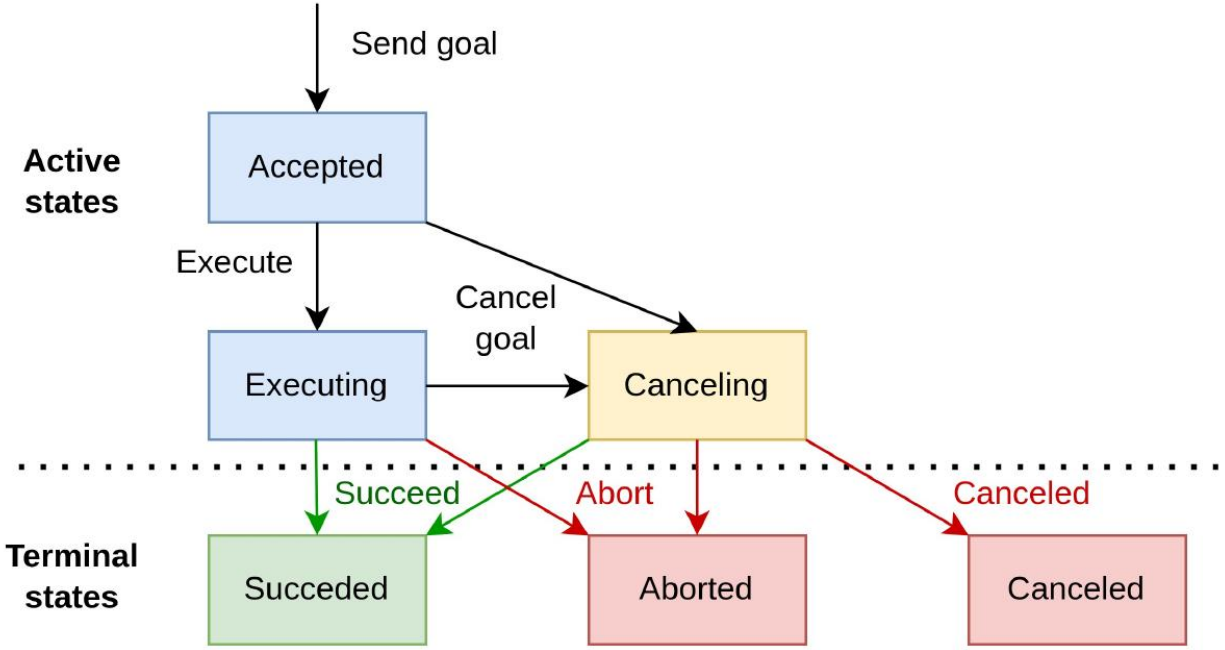
ROS 2 Actions

- Actions are built on *topics* and *services*
- Their functionality is similar to services, except **actions can be canceled**. They also provide steady **feedback**, as opposed to services which return a single response (essential for long-running robotic tasks).
- Actions require **more code** than topics or services but offer more control



ROS 2 Actions

- Any time an action server receives a goal from a client, it can decide if accepting or rejecting it.
- If accepting it, the server creates a new state machine for the goal:



State	Description / Transition
`accepted` → `executing`	Goal accepted, action running
`executing` → `succeeded`	Task completed successfully
`executing` → `aborted`	Task failed
`executing` → `canceled`	Client requested cancel

Custom ROS 2 Actions

- As for messages and services, also action shave a specific format (.action) and standard destination folder (*action)*.

- A *request* message is sent from an action client to an action server initiating a new goal.
- A *result* message is sent from an action server to an action client when a goal is done.
- *Feedback* messages are periodically sent from an action server to an action client with updates about a goal.

```
# Request
---
# Result
---
# Feedback
```

Custom ROS 2 Actions

- Let's compute the Fibonacci sequence!
- Create an *action* directory in your custom msgs package and then the file Fibonacci.action:
  mkdir action
  touch Fibonacci.action
- The goal request is the *order* of the Fibonacci sequence we want to compute, the result is the final *sequence*, and the feedback is the *partial_sequence* computed so far

```
int32 order
---
int32[] sequence
---
int32[] partial_sequence
```

Compile the Custom Action

- In the *CmakeLists.txt:*

```
find_package(rosidl_default_generators REQUIRED)

rosidl_generate_interfaces(${PROJECT_NAME}
 "action/Fibonacci.action“
)
```

- In the *package.xml:*

```
<buildtool_depend>rosidl_default_generators</buildtool_depend>
<depend>action_msgs</depend>
<member_of_group>rosidl_interface_packages</member_of_group>
```

Test the Custom Action

- After we compiled, test from the command line:

```
# Source our workspace
. install/setup.bash

# Check that our action definition exists
ros2 interface show <action_pkg_name>/action/Fibonacci
```

Writing the Action Server

An action server requires 6 things:

1. The templated action type name: *Fibonacci*.
2. A ROS 2 node to add the action to: *this*.
3. The action name: *'fibonacci'*.
4. A callback function for handling goals: *handle_goal.*
5. A callback function for handling cancellation: *handle_cancel*.
6. A callback function for handling goal acceptance: *handle_accept*.

Writing the Action Server

```cpp
#include "rclcpp/rclcpp.hpp"
#include "rclcpp_action/rclcpp_action.hpp"

#include "calculator_msgs/action/fibonacci.hpp"

class FibonacciActionServer : public rclcpp::Node
{
public:
  using Fibonacci = calculator_msgs::action::Fibonacci;
  using GoalHandleFibonacci = rclcpp_action::ServerGoalHandle<Fibonacci>;

  FibonacciActionServer(const rclcpp::NodeOptions & options = rclcpp::NodeOptions())
    : Node("fibonacci_action_server_node", options){ }

private:
  rclcpp_action::Server<Fibonacci>::SharedPtr action_server_;


};  // class FibonacciActionServer
```

```cpp
int main(int argc, char * argv[])
{
  rclcpp::init(argc, argv);
  rclcpp::spin(std::make_shared<FibonacciActionServer>());
  rclcpp::shutdown();
  return 0;
}
```

Writing the Action Server

```
class FibonacciActionServer : public rclcpp::Node
{
public:
  using Fibonacci = calculator_msgs::action::Fibonacci;
  using GoalHandleFibonacci = rclcpp_action::ServerGoalHandle<Fibonacci>;

  FibonacciActionServer(const rclcpp::NodeOptions & options = rclcpp::NodeOptions()): Node("fibonacci_action_server_node", options)
  {
    using namespace std::placeholders;

    this->action_server_ = rclcpp_action::create_server<Fibonacci>(
      this,
      "fibonacci",
      std::bind(&FibonacciActionServer::handle_goal, this, _1, _2),
      std::bind(&FibonacciActionServer::handle_cancel, this, _1),
      std::bind(&FibonacciActionServer::handle_accepted, this, _1));}
```

Writing the Action Server

```
  // Callback for handling new goals
 rclcpp_action::GoalResponse handle_goal(
  const rclcpp_action::GoalUUID & uuid,
  std::shared_ptr<const Fibonacci::Goal> goal)
 {
  RCLCPP_INFO(this->get_logger(), "Received goal request with order %d", goal-
>order);
  (void)uuid;
  return rclcpp_action::GoalResponse::ACCEPT_AND_EXECUTE;
 }

 // Callback for dealing with cancellation requests
 rclcpp_action::CancelResponse handle_cancel(
  const std::shared_ptr<GoalHandleFibonacci> goal_handle)
 {
  RCLCPP_INFO(this->get_logger(), "Received request to cancel goal");
  (void)goal_handle;
  return rclcpp_action::CancelResponse::ACCEPT;}
```

```
  // Callback for handling accepted goals and processing them.
  // Since the execution is a long-running operation, we spawn off a
thread to do the actual work and return from handle_accepted quickly
  void handle_accepted(const std::shared_ptr<GoalHandleFibonacci>
goal_handle)
  {
   using namespace std::placeholders;
   // this needs to return quickly to avoid blocking the executor, so spin
up a new thread
   std::thread{std::bind(&FibonacciActionServer::execute, this, _1),
goal_handle}.detach();
  }
```

Writing the Action Server

```
  // This work thread processes one sequence number of the Fibonacci sequence every second,
  // publishing a feedback update for each step. When it has finished processing,
  // it marks the goal_handle as succeeded, and quits.
  void execute(const std::shared_ptr<GoalHandleFibonacci> goal_handle)
  {
    RCLCPP_INFO(this->get_logger(), "Executing goal");
    rclcpp::Rate loop_rate(1);
    const auto goal = goal_handle->get_goal();
    auto feedback = std::make_shared<Fibonacci::Feedback>();
    auto & sequence = feedback->partial_sequence;
    sequence.push_back(0);
    sequence.push_back(1);
    auto result = std::make_shared<Fibonacci::Result>();
```

```
    for (int i = 1; (i < goal->order) && rclcpp::ok(); ++i) {
      // Check if there is a cancel request
      if (goal_handle->is_canceling()) {
        result->sequence = sequence;
        goal_handle->canceled(result);
        RCLCPP_INFO(this->get_logger(), "Goal canceled");
        return;
      }

      // Update sequence
      sequence.push_back(sequence[i] + sequence[i - 1]);
      // Publish feedback
      goal_handle->publish_feedback(feedback);
      RCLCPP_INFO(this->get_logger(), "Publish feedback");

      loop_rate.sleep();
    }

    // Check if goal is done
    if (rclcpp::ok()) {
      result->sequence = sequence;
      goal_handle->succeed(result);
      RCLCPP_INFO(this->get_logger(), "Goal succeeded");}
  }
```

Compiling the Action Server

In the CMakeLists.txt, add:

```
find_package(rclcpp_action REQUIRED)

add_executable(fibonacci_action_server src/fibonacci_action_server.cpp)
ament_target_dependencies(fibonacci_action_server rclcpp rclcpp_action
<action_pkg_name>)

install(TARGETS
  fibonacci_action_server
  DESTINATION lib/${PROJECT_NAME}
)
```

Running the Action Server

- After building:

```
# Source our workspace
. install/setup.bash

# Run the server node
ros2 run <server_pkg_name> fibonacci_action_server

# Verify from the terminal
ros2 action send_goal /fibonacci <action_pkg_name>/action/Fibonacci order:\ 7\

ros2 action send_goal --feedback /fibonacci <action_pkg_name>/action/Fibonacci
order:\ 7\
```

Writing the Action Client

An action client requires 3 things:

1. The templated action type name: *Fibonacci*.
2. A ROS 2 node to add the action client to: *this*.
3. The action name: *'fibonacci'*.
4. (Optional) Response, feedback, and result callbacks.

Writing the Action Client

```cpp
#include "rclcpp/rclcpp.hpp"
#include "rclcpp_action/rclcpp_action.hpp"

#include "calculator_msgs/action/fibonacci.hpp"

class FibonacciActionClient : public rclcpp::Node
{
public:
  using Fibonacci = calculator_msgs::action::Fibonacci;
  using GoalHandleFibonacci = rclcpp_action::ClientGoalHandle<Fibonacci>;

  FibonacciActionClient(const rclcpp::NodeOptions & options = rclcpp::NodeOptions())
  : Node("fibonacci_action_client_node", options){  }

private:
  rclcpp_action::Client<Fibonacci>::SharedPtr client_ptr_;
  rclcpp::TimerBase::SharedPtr timer_;
};  // class FibonacciActionClient
```

```cpp
int main(int argc, char * argv[])
{
  rclcpp::init(argc, argv);
  rclcpp::spin(std::make_shared<FibonacciActionClient>());
  rclcpp::shutdown();
  return 0;
}
```

Writing the Action Client

```
class FibonacciActionClient : public rclcpp::Node
{
public:
  using Fibonacci = calculator_msgs::action::Fibonacci;
  using GoalHandleFibonacci = rclcpp_action::ClientGoalHandle<Fibonacci>;

  FibonacciActionClient(const rclcpp::NodeOptions & options = rclcpp::NodeOptions())
  : Node("fibonacci_action_client_node", options)
  {
    this->client_ptr_ = rclcpp_action::create_client<Fibonacci>(this, "fibonacci");

    this->timer_ = this->create_wall_timer(
      std::chrono::milliseconds(500),
      std::bind(&FibonacciActionClient::send_goal, this));
  }
```

Writing the Action Client

```
void send_goal()  {
  using namespace std::placeholders;

  // Cancels the timer (so the function is only called once)
  this->timer_->cancel();

  // Waits for the action server to come up
  if (!this->client_ptr_->wait_for_action_server()) {
    RCLCPP_ERROR(this->get_logger(), "Action server not available after waiting");
    rclcpp::shutdown();
  }

  // Instantiates a new Fibonacci::Goal
  auto goal_msg = Fibonacci::Goal();
  goal_msg.order = 10;
  RCLCPP_INFO(this->get_logger(), "Sending goal");

  // Sets the response, feedback, and result callbacks
   auto send_goal_options =
rclcpp_action::Client<Fibonacci>::SendGoalOptions();
   send_goal_options.goal_response_callback =
    std::bind(&FibonacciActionClient::goal_response_callback, this, _1);
   send_goal_options.feedback_callback =
    std::bind(&FibonacciActionClient::feedback_callback, this, _1, _2);
   send_goal_options.result_callback =
    std::bind(&FibonacciActionClient::result_callback, this, _1);
   //Sends the goal to the server
   this->client_ptr_->async_send_goal(goal_msg, send_goal_options);
   // this->client_ptr_->async_send_goal(goal_msg);
  }
```

Writing the Action Client

```
private:
 rclcpp_action::Client<Fibonacci>::SharedPtr client_ptr_;
 rclcpp::TimerBase::SharedPtr timer_;

 void goal_response_callback(const GoalHandleFibonacci::SharedPtr & goal_handle){
  if (!goal_handle) {
   RCLCPP_ERROR(this->get_logger(), "Goal was rejected by server");
  } else {
   RCLCPP_INFO(this->get_logger(), "Goal accepted by server, waiting for result"); }

 void feedback_callback(
  GoalHandleFibonacci::SharedPtr,
  const std::shared_ptr<const Fibonacci::Feedback> feedback){
  std::stringstream ss;
  ss << "Next number in sequence received: ";
  for (auto number : feedback->partial_sequence) {
   ss << number << " ";}
  RCLCPP_INFO(this->get_logger(), ss.str().c_str());}
```

```
void result_callback(const GoalHandleFibonacci::WrappedResult & result)
{
 switch (result.code) {
  case rclcpp_action::ResultCode::SUCCEEDED:
   break;
  case rclcpp_action::ResultCode::ABORTED:
   RCLCPP_ERROR(this->get_logger(), "Goal was aborted");
   return;
  case rclcpp_action::ResultCode::CANCELED:
   RCLCPP_ERROR(this->get_logger(), "Goal was canceled");
   return;
  default:
   RCLCPP_ERROR(this->get_logger(), "Unknown result code");
   return;
 }
 std::stringstream ss;
 ss << "Result received: ";
 for (auto number : result.result->sequence) {
  ss << number << " ";
 }
 RCLCPP_INFO(this->get_logger(), ss.str().c_str());
 rclcpp::shutdown();
}
```

Compiling the Action Client

In the CMakeLists.txt, add:

```
add_executable(fibonacci_action_client src/fibonacci_action_client.cpp)
ament_target_dependencies(fibonacci_action_clientrclcpp rclcpp_action
<action_pkg_name>)

install(TARGETS
  fibonacci_action_server
  fibonacci_action_client
  DESTINATION lib/${PROJECT_NAME}
)
```

Running the Action Client

After building:

```
# Source our workspace
. install/setup.bash

# Run the server node
ros2 run <server_pkg_name> fibonacci_action_server

# Run the client node
ros2 run <server_pkg_name> fibonacci_action_client
```

UNIVERSITÀ
DI TRENTO
Department of
Information Engineering and Computer Science

ROS 2 Tools & Simulators

Edoardo Lamon

Software Development for Collaborative Robots

Academic Year 2025/26

ROS 2 Tools

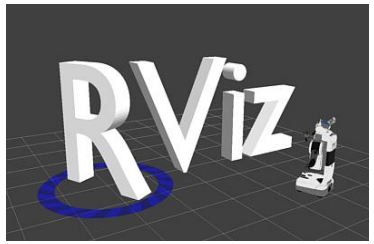
Transform and RViz

The Tools

- We will look at rviz (2) and Gazebo/CoppeliaSim

> *"rviz shows you what the robot **thinks** it's happening, while Gazebo (CopelliaSim) shows you what is **really** happening."*
>
> *Morgan Quigley*

- Gazebo and CopelliaSim are **physics simulators**
- Rviz is a visualization tool which enables the user to see the **robot's state** and **perception**



tf2 and rviz

- First install the needed packages

  sudo apt-get install ros-humble-turtle-tf2-py ros-humble-tf2-tools ros-humble-tf-transformations
- Then
  - Launch this particular turtle simulation

    ros2 launch turtle_tf2_py turtle_tf2_demo.launch.py
  - In another terminal launch the controller

    ros2 run turtlesim turtle_teleop_key

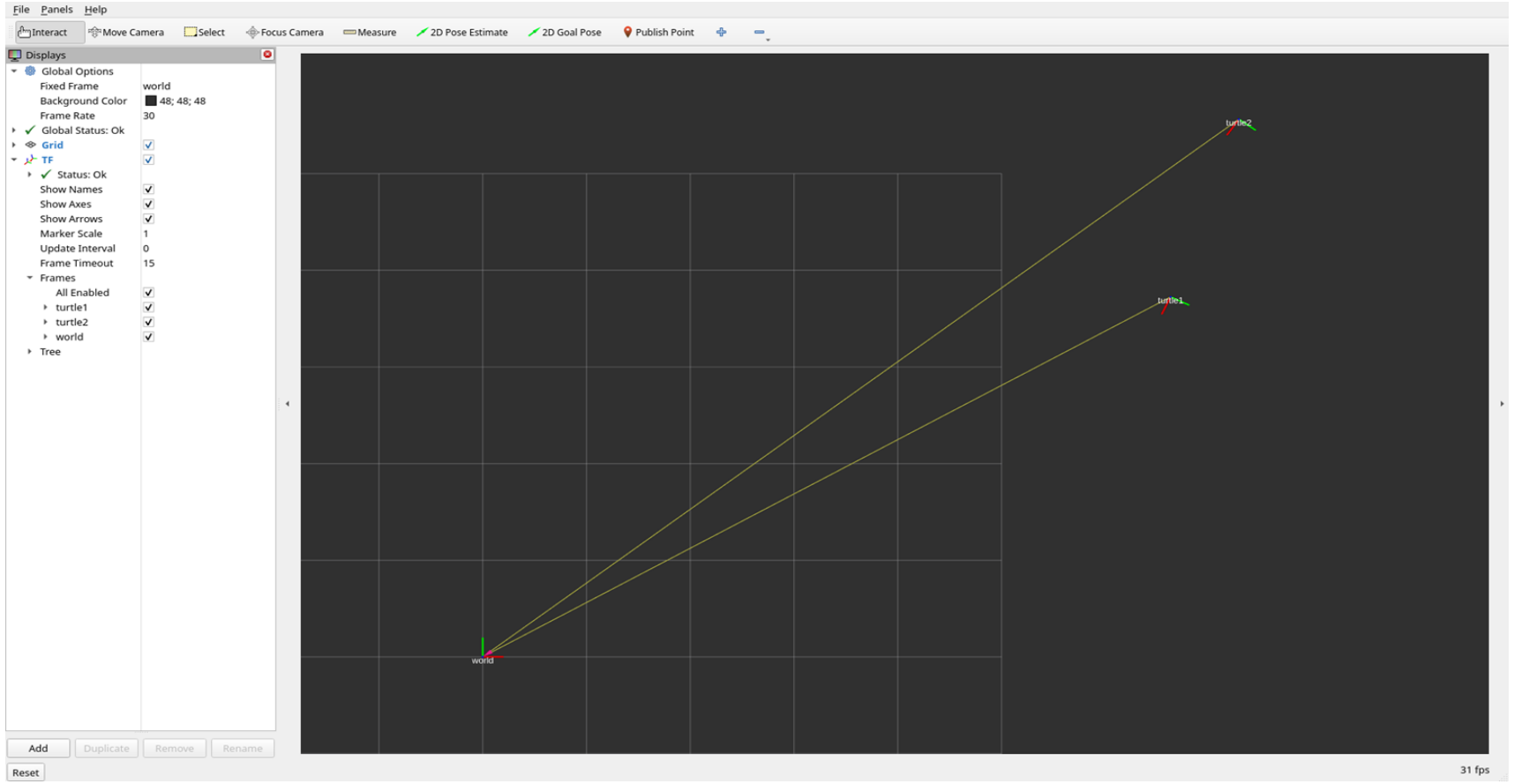
  This demo is using the tf2 library to create three coordinate frames: a world frame, a turtle1 frame, and a turtle2 frame. This tutorial uses a tf2 broadcaster to publish the turtle coordinate frames and a tf2 listener to compute the difference in the turtle frames and move one turtle to follow the other.

- Open rviz, a visualization tool that is useful for examining tf2 frames:

  ros2 run rviz2 rviz2 -d $(ros2 pkg prefix --share turtle_tf2_py)/rviz/turtle_rviz.rviz

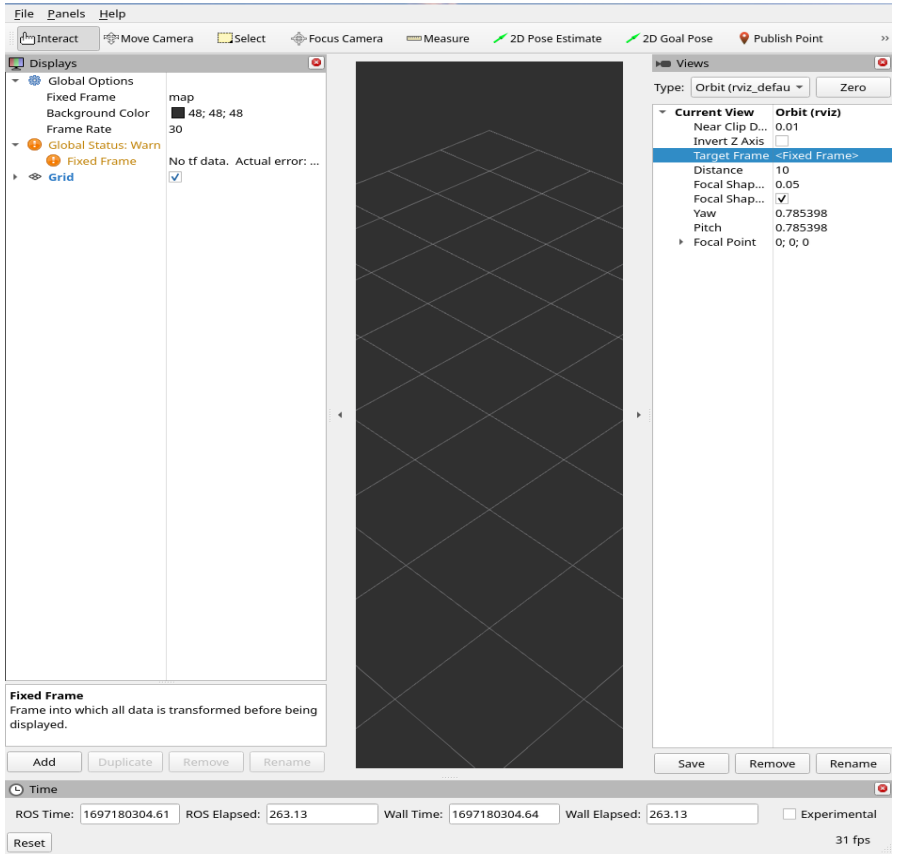
tf2 and rviz

In the side bar you will see the **coordinate frames** broadcasted by tf2.
As you drive the turtle around you will see the frames move in rviz.



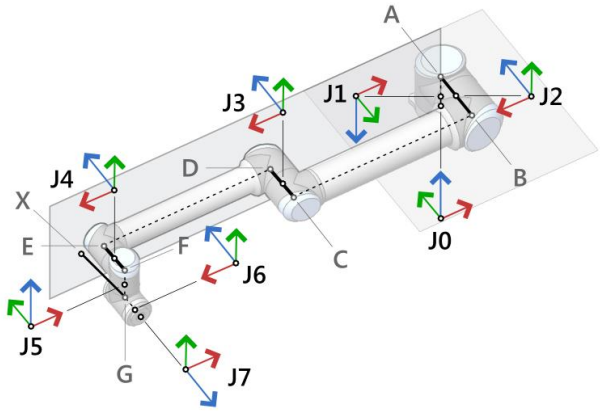
rviz

- It has two frames controlling how data is displayed:
  - **Fixed frame**: is the frame all incoming data is *transformed into* before being displayed, hence it should be set to either a *root element* (like map) or a *fixed frame*
  - **Target frame**: reference frame for the camera view
- "Pose estimate" used to initialize the position of your robot → sends the position on /initialpose



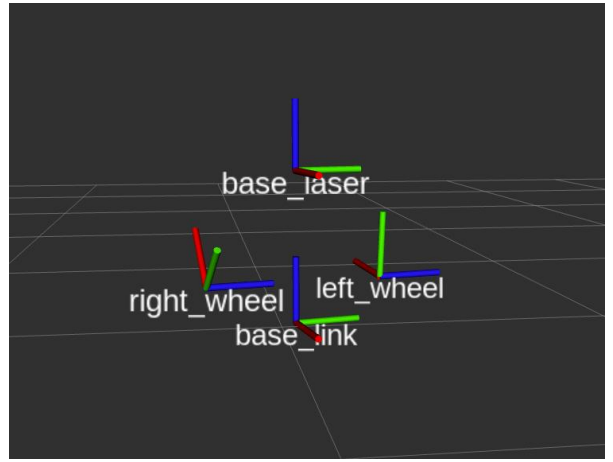
tf2

- We are looking at **coordinate frames**
- tf2 is a special library of ROS2 which publishes the **transformation matrices** between coordinates frames in the topic /tf2
- Necessary to easily understand the position of one coordinate frame w.r.t. another



UR5 coordinate frames

Mobile robot's coordinate frames



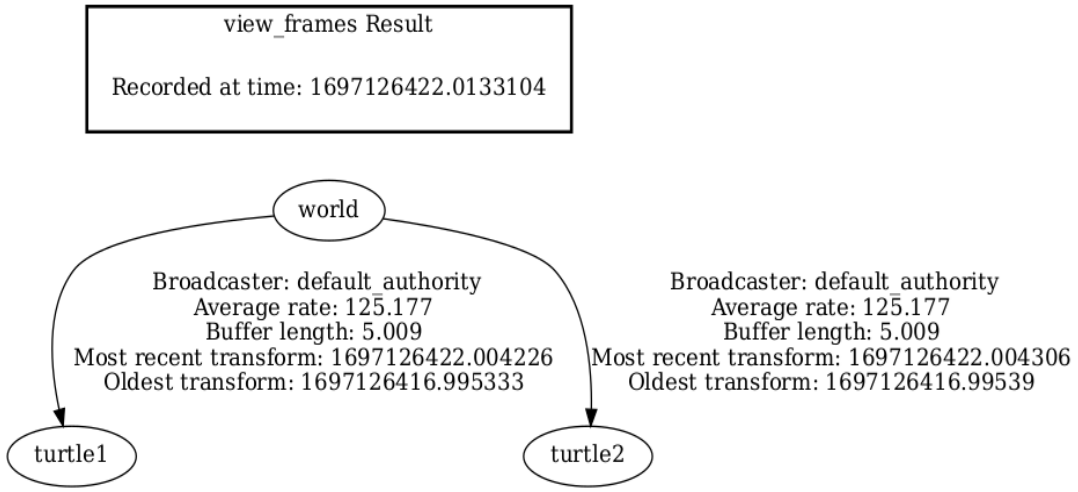
tf2

- If using wsl, install wslview
  sudo apt install wslu
- We can see the relations between frames using

  ros2 run tf2_tools view_frames

  wslview <name_of_pdf>
- We can look at the transform between two frames by running

  ros2 run tf2_ros tf2_echo [source_frame] [target_frame]



tf2

With tf2 you can:

- Define static broadcaster (define the relationship between a robot base and fixed sensors or non-moving parts);
- Define a broadcaster (define the relationship between a robot base and moving parts → timestamped transformations);
- Define a listener (to use the published transformation in a code)
- And more (s.ee https://docs.ros.org/en/humble/Tutorials/Intermediate/Tf2/Tf2-Main.html)

ROS 2 Tools

URDF

URDF

- rviz2 uses **Unified Robot Description Format (URDF)** for robot models → XML format
- We have to specify:
  - *Robot*: information on the robot
  - *Links*: the components of the robots
  - *Joints*: the interactions between links
  - Many more, see https://docs.ros.org/en/humble/Tutorials/Intermediate/URDF/URDF-Main.html#

```
<?xml version="1.0"?>
<robot name="myfirst">
 <link name="base_link">
   <visual>
     <geometry>
       <cylinder length="0.6" radius="0.2"/>
     </geometry>
   </visual>
 </link>
</robot>
```

URDF - Links

A **link** describes a rigid body and may have the following properties:

- **Visual**: how the body **should appear** in the simulation. There may be *more than one* and together they represent the body. Within visual you specify:
  - *Origin*: the reference frame of the visual w.r.t. the reference frame of the link → expressed as **offset**
  - *Geometry*: the shape, which may be: box, sphere, cylinder, mesh
  - *Material*: the material of the visual element
- **Collision**: similar to visual, but used to check for collision during simulation
  - May not have the same shape → easier check on collisions and safer zones
- **Inertial**: define some physical properties of the robot for the simulation
  - *Inertia*: rotational inertia matrix → mandatory
  - *Mass*: in kilograms

URDF - Joints

- A **joint** describes how two links interact:
- When defining, we must specify the **type**:
  - **fixed**: the joint cannot move
  - **revolute**: it rotates *along* the axis and we *must limit* the range with upper and lower limits.
  - **continuous**: a continuous hinge joint that rotates around the axis and *has no* upper and lower limits.
  - **prismatic**: a sliding joint that slides along the axis, and *has a limited* range specified by the upper and lower limits.
  - **floating**: this joint allows motion for all 6 degrees of freedom.
  - **planar**: this joint allows motion in a plane perpendicular to the axis.
- The elements inside the joint may be:
  - **Origin**: represent a transform from the parent link to the child link
  - **Parent**: the parent link
  - **Child**: the child link
  - **Limit**: the limits to be respected when using type revolute or prismatic
  - See reference for more

Visualizing an URDF

Install the dependency:

```
sudo apt install ros-humble-urdf-tutorial -y
```

URDF models are usually placed in the *urdf* folder. We will visualize now with:

```
ros2 launch urdf_tutorial display.launch.py model:=urdf/<robot.urdf>
```

This launch does three things:

- Loads the specified model and saves it as a parameter for the *robot_state_publisher* node;
- Runs nodes to publish sensor_msgs/msg/JointState and transforms;
- Starts Rviz with a configuration file.

Visualizing an URDF

Example URDFs are located in urdf_tutorial:

- Inspect the URDF file:

code /opt/ros/humble/share/urdf_tutorial/urdf/<robot.urdf>

- Visualize the robot in the URDF file in rviz:

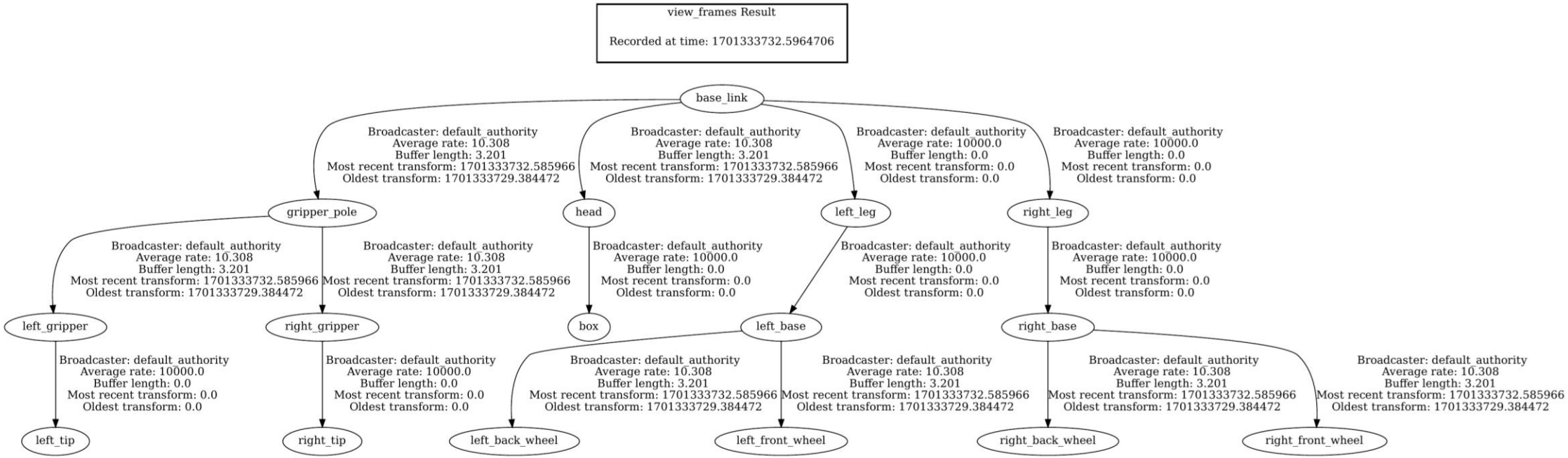
ros2 launch urdf_tutorial display.launch.py model:=urdf/<robot.urdf>

Examples:

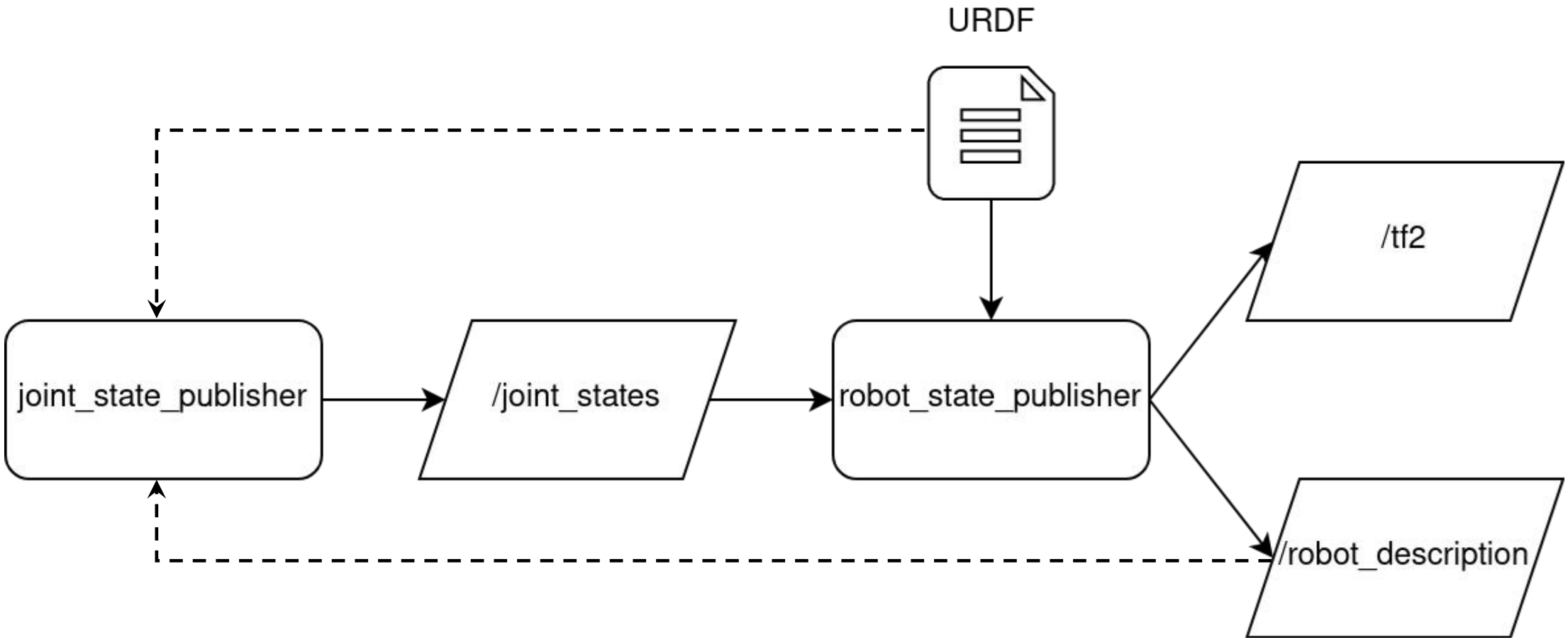
- Single link: 01-myfirst.urdf
- Two links with a fixed joint: 02-multipleshapes.urdf; 03-origins.urdf (specifies also where the second shape is originated)
- Three links of specific material/color with two fixed joints: 04-materials.urdf
- Multiple links and joints: 05-visual.urdf; 06-flexible.urdf (joints are flexible, with position, velocity or effort limits)
- Links with collision and inertial properties: 07-physics.urdf

Check the Tree Frames

ros2 run tf2_tools view_frames



Joint State and Robot State



Xacro and URDF

- ROS2 allows for using *xacro*, a macro language for XML (.xacro)
- This files use **macros** to ease some aspects of URDF files:
  - **constants** so to not have to change the same numeric value in many places;
  - **mathematical operations** to compute values;
  - **macros** to define whole pieces of code, also *parametrized*.

Xacro and URDF

It is possible to use xacro in 2 ways:

- To compile from .xacro to .urdf use

  xacro file.xacro -o file.urdf

- Automatically generate the urdf in a launch file. This is convenient because it stays **up to date** and doesn't use up hard drive space. However, it does take time to generate, so be aware that your launch file might take longer to start up.

  ros2 launch urdf_tutorial display.launch.py model:=urdf/08-macroed.urdf.xacro

Visualizing an URDF

urdf_launch, uses

- joint_state_publisher, which, will continuously publish all joint states to /joint_states. It reads the description of the robot by:
  - listening to /robot_description;
  - an input URDF.
- robot_state_publisher, which publishes the state of a robot to tf2 by reading the URDF and by subscribing to joint_state_publisher to get individual joint states

Visualizing an URDF

We can use the package *urdf_launch* to:

- **Load Robot description** (description.launch.py):
  - Loads the URDF/xacro robot model as a parameter, based on launch arguments;
  - Launches a single node, *robot_state_publisher*, with the robot model parameter;
  - *Robot description* becomes available as a topic.

```
def generate_launch_description():
ld = LaunchDescription()
ld.add_action(IncludeLaunchDescription(PathJoinSubstitution([FindPackageShare('urdf_launch'), 'launch', 'description.launch.py']),
   launch_arguments={
    'urdf_package': 'urdf_tutorial',
    'urdf_package_path': PathJoinSubstitution('urdf', '06-flexible.urdf'])}.items() ))
return ld
```

Visualizing an URDF

We can use the package *urdf_launch* to:

- **Display Robot Model (**display.launch.py**):**
  - Display just the robot model in Rviz with a preconfigured setup;
  - Launches a joint state publisher (with optional GUI), which publishes in the topic /joint_states msgs of the type sensor_msgs::msg::JointState.

```
def generate_launch_description():
ld = LaunchDescription()
ld.add_action(IncludeLaunchDescription(PathJoinSubstitution([FindPackageShare('urdf_launch'), 'launch', 'display.launch.py']),
   launch_arguments={
    'urdf_package': 'urdf_tutorial',
    'urdf_package_path': PathJoinSubstitution(‘urdf', '06-flexible.urdf'])}.items() ))
return ld
```

Execise

- Clone the UR Description repo and compile the workspace:
  git clone -b humble https://github.com/UniversalRobots/Universal_Robots_ROS2_Description.git ur_description
- Inspect *ur.urdf.xacro* and *ur.macro.xacro* the urdf folder. All the UR model URDFs can be generated using the same xacro files and the data in the config folder;
- Generate the URDF of the UR5e in a launch file and visualize it.

- Display the TF and their names;
- Modify the joint configuration; what happens with elbow_joint:=2.992? Why is happening?

ROS 2 Tools

Simulators

Physics Simulators

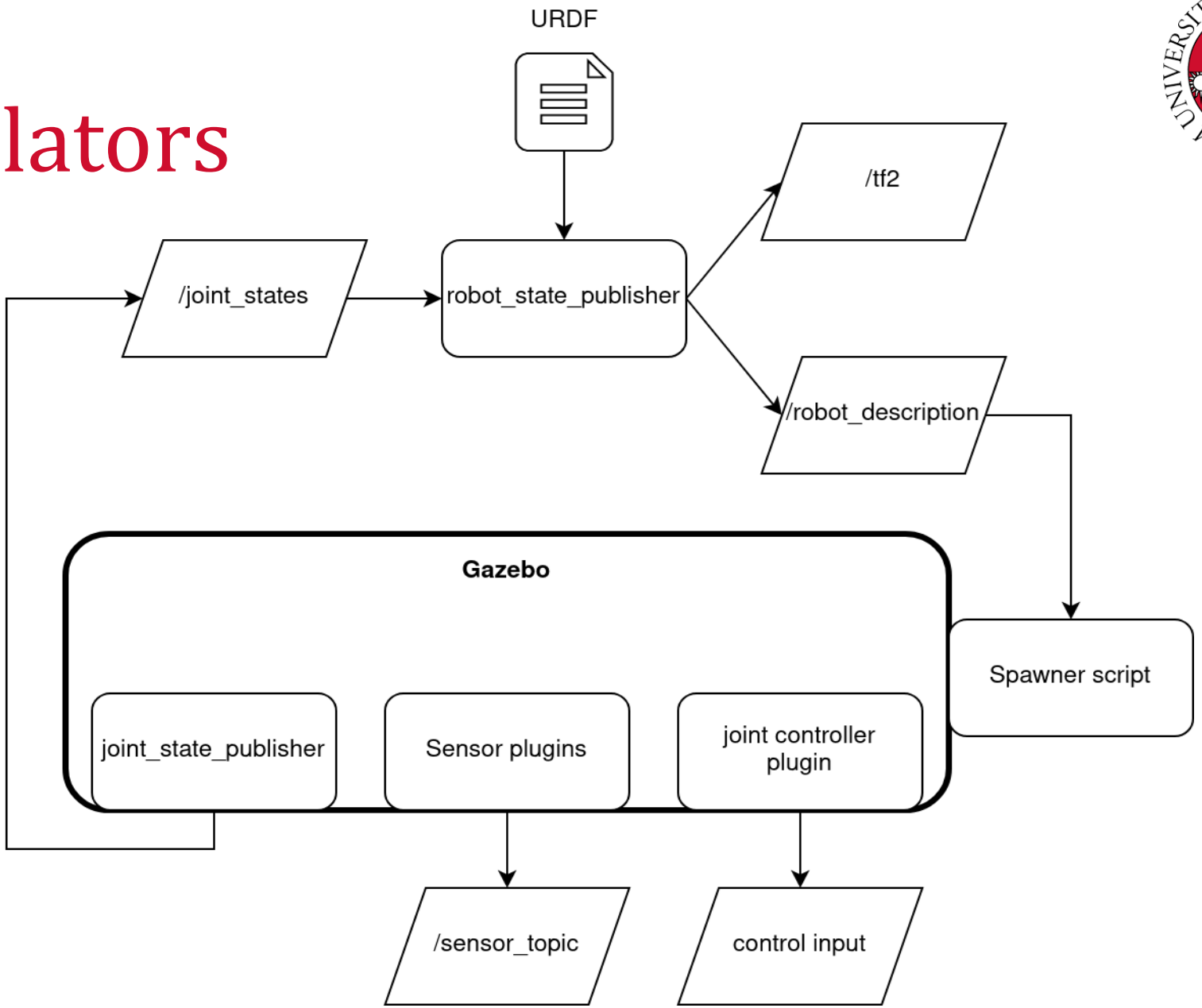
- Why is happening? Because **in rviz physics is not simulated** (it is a **visualization tool**), so one body can pass through another.
- We could manually publish the **joint states**, but this is not the task of the robot programmer → On a real robot this is **read from the sensors on the joint of the robot.**
- Before testing on a real robot, one can use a **physics engine** to get the joint states (here is a benchmark):
  - Open Dynamics Engine (ODE);
  - Dynamic Animation and Robotics Toolkit (DART);
  - Bullet;
  - Multi-Joint dynamics with Contact (MuJoCo);
  - And many more…

Physics Simulators

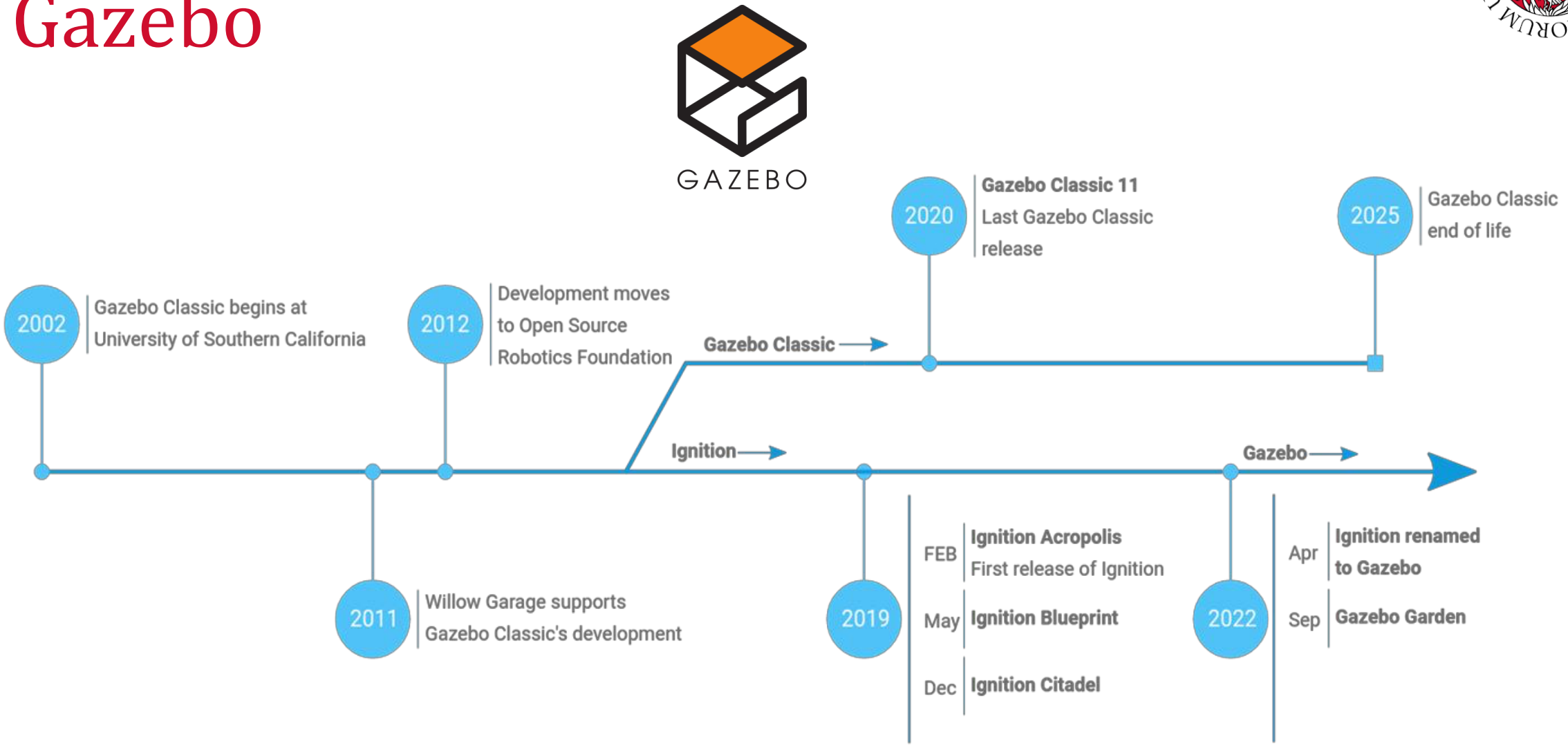
Usually, these general physics simulators are used in **robotic physics simulators** which allows simple integration with ROS (another benchmark):

- **Gazebo**: open-source from the OSRF with support for ROS2
- **CoppeliaSim**: free-educational with support for ROS2 and Python/Lua support for scripts in CoppeliaSim, lighweight
- **Unity**: free-educational with support for ROS2
- **Webots**: open-source with support for ROS2
- **NVIDIA Omniverse**/**Isaac Sim**: free for individuals, with ROS2 bridge, very demanding system requirements

Physics Simulators



Gazebo


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

- The recommended version of Gazebo with ROS 2 Humble is GZ Fortress (list of features);
  sudo apt install ros-humble-ros-gz

- Launch a demo simulation:
  ign gazebo -v 4 -r visualize_lidar.sdf
  if it crashes, try with:
  ign gazebo -v 4 -r visualize_lidar.sdf --render-engine ogre

- Check gazebo topics: ign topic -l
- Check ros2 topics (gazebo and ros2 are separated)

Gazebo

- Install a bridge for the topics and the keyboard teleop package:

  sudo apt-get install ros-humble-ros-ign-bridge ros-humble-teleop-twist-keyboard

- We can map the ROS topic in a Gazebo topic:

  ros2 run ros_gz_bridge parameter_bridge /model/vehicle_blue/cmd_vel@geometry_msgs/msg/Twist]ignition.msgs.Twist

- And finally, teleoperate the blue robot:

  ros2 run teleop_twist_keyboard teleop_twist_keyboard --ros-args -r /cmd_vel:=/model/vehicle_blue/cmd_vel

Gazebo

- Gazebo uses **.sdf** (Simulation Description Format) as model files, an XML format similar to URDF, but:
  - Describe also the **world** and not only the models;
  - Use **plugins** to describe how to interact with other programs, such as ROS.
- Gazebo concepts can be found here;
- Robot models described by URDF can also be spawned in the simulation environment:

```
ign gazebo empty.sdf
ign service -s /world/empty/create --reqtype ignition.msgs.EntityFactory --reptype
ignition.msgs.Boolean --timeout 1000 --req 'sdf_filename:
"/opt/ros/humble/share/urdf_tutorial/urdf/07-physics.urdf", name: "urdf_model"'
```

Gazebo and Control

The robot controller can be directly written as a Gazebo plugin (shared library using Gazebo API), independent from ROS 2, but this has many limits:

- You **lose the standard interfaces** (joint state broadcaster, controller_manager, ros2 control CLI).
- Many ROS 2 packages (MoveIt, Nav2, tools expecting hardware_interface) **assume ros2_control**; without it, integration gets harder.
- You'll need to maintain your **own message schema**, plugins, and lifecycle/timeout/latency handling.

```
<gazebo>
 <plugin name="gazebo_ros_diff_drive"
      filename="libgazebo_ros_diff_drive.so">
  <left_joint>chassis_to_left_wheel_joint</left_joint>
  <right_joint>chassis_to_right_wheel_joint</right_joint>
  <wheel_separation>${body_width+0.04}</wheel_separation>
  <wheel_diameter>${wheel_radius*2.0}</wheel_diameter>

  <max_wheel_torque>200</max_wheel_torque>
  <max_wheel_accelaration>10</max_wheel_accelaration>

  <odometry_frame>odom</odometry_frame>
  <robot_base_frame>base_link</robot_base_frame>
  <publish_odom>true</publish_odom>
  <publish_wheel_tf>true</publish_wheel_tf>
  <publish_odom_tf>true</publish_odom_tf>
 </plugin>
</gazebo>
```

In this course we will use **ros2_control**, which provides scalable and reusable implementation of standard controllers that can work on **real hardware**!

CoppeliaSim

- Different OS supported;
- Low CPU usage;
- Different physics simulator integrated:
  - Bullet;
  - ODE;
  - Vortex;
  - Newton
- Comparison studies available:
  - https://www.sciencedirect.com/science/article/pii/S1569190X22001046
  - https://arxiv.org/pdf/2204.06433

CoppeliaSim

- Download from https://www.coppeliarobotics.com/downloads:

wget https://downloads.coppeliarobotics.com/V4_8_0_rev0/CoppeliaSim_Edu_V4_10_0_rev0_Ubuntu22_04.tar.xz

- Extract it:

tar -xf CoppeliaSim_Edu_V4_10_0_rev0_Ubuntu22_04.tar.xz4

- Install dependencies:

sudo apt update; sudo apt install xsltproc; python3 -m pip install pyzmq cbor2 xmlschema

- To run the application:

cd CoppeliaSim_Edu_V4_10_0_rev0_Ubuntu22_04

./coppeliaSim.sh

In the case of a newer release, please update the path/name to the file accordingly.

External controller with CoppeliaSim

CoppeliaSim provides several ways to specify a robot controller:

- With a **simulation script** (in Python or Lua), or writing a **plugin** (in a language able to generate a shared library and able to call exported C-functions), or calling the **client libraries** (C++/Python);
- Using the **remote API**, such as the ZeroMQ remote API. In this way, you can run the control code from an external application, from a robot or from another computer. This also allows you to control a simulation with the exact same code as the one that runs the real robot.
- Using a the **ROS2 Interface**, which is available in the simROS2 package. The interface will allow the creation of ROS2 Communication Interfaces (topic, services, etc.) that will be directly available in ROS.

CoppeliaSim – ROS2 Bridge

- CoppeliaSim is a general-purpose robotic simulator and does not require ROS to work;
- You can program directly the robot control/motion planner etc in CoppeliaSim using the Python/Lua scripts..
- But if we do this, we will lose the modularity of the ROS framework and the use of ROS tools!
- So, we will need a bridge between ROS2 and CoppeliaSim→ CoppeliaSim provides one in
  CoppeliaSim_{version}/programming/ros2_packages

CoppeliaSim – ROS2 Bridge

- In CoppeliaSim_{version}/programming/ros2_packages 2 ROS2 packages are available:
  - sim_ros2_interface;
  - ros2_bubble_rob.
- You can copy them into your workspace and then compile it:

```
$ export COPPELIASIM_ROOT_DIR=~/path/to/coppeliaSim/folder
$ ulimit -s unlimited #otherwise compilation might freeze/crash
$ colcon build --symlink-install --cmake-args -DCMAKE_BUILD_TYPE=Release
```

- Let's follow the tutorial to run the *bubble_rob* example: https://www.coppeliarobotics.com/helpFiles/en/ros2Tutorial.htm

UNIVERSITÀ
DI TRENTO
Department of
Information Engineering and Computer Science

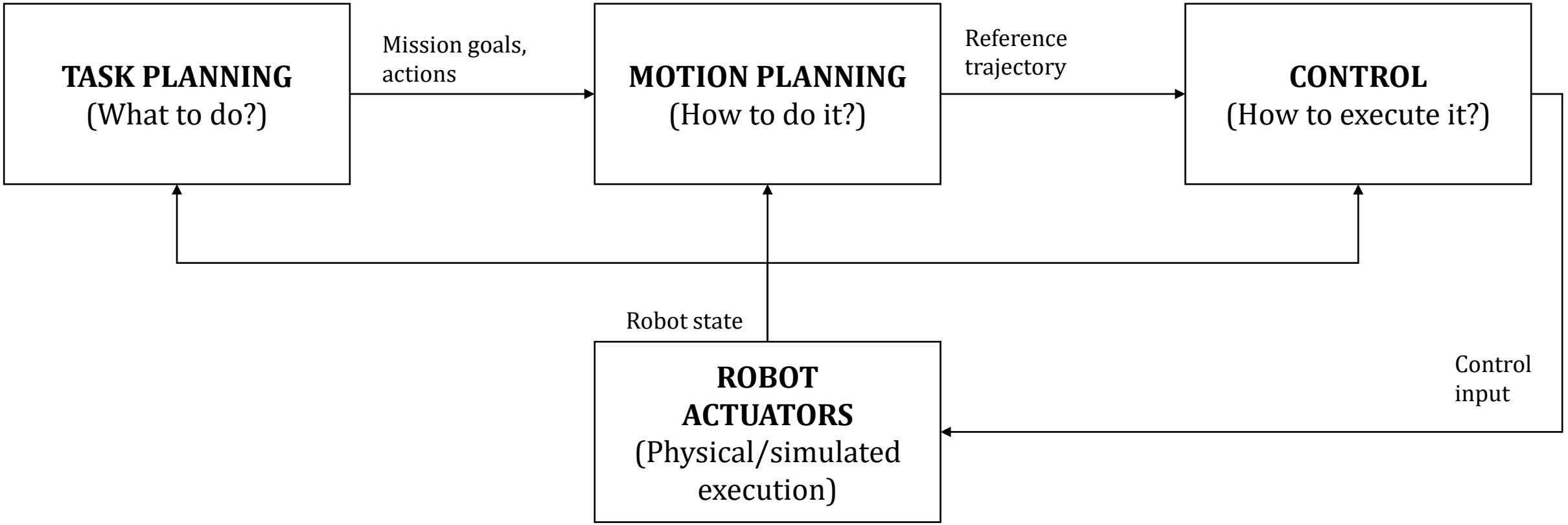
ROS 2 Control

Edoardo Lamon

Software Development for Collaborative Robots

Academic Year 2025/26

High-level Framework for a Robotic Application

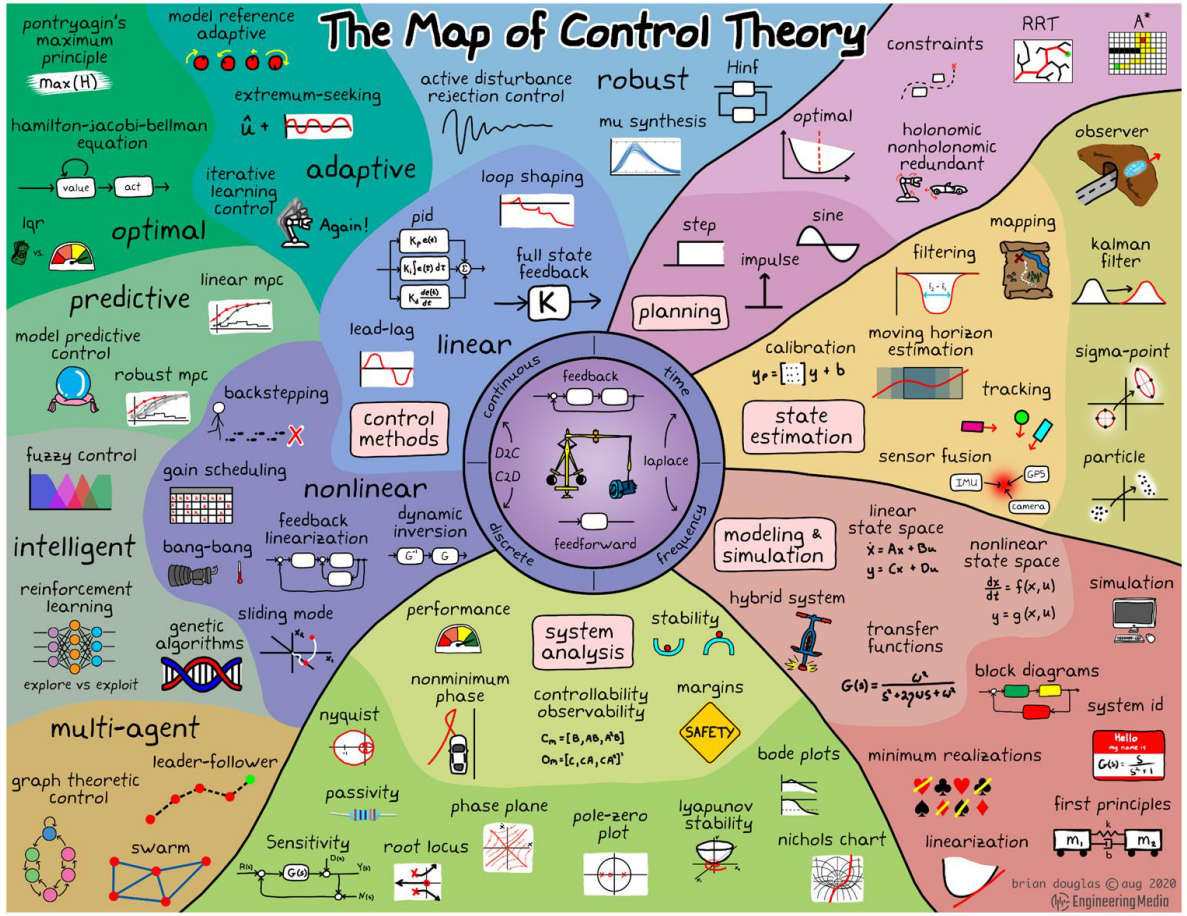


High-level Framework for a Robotic Application

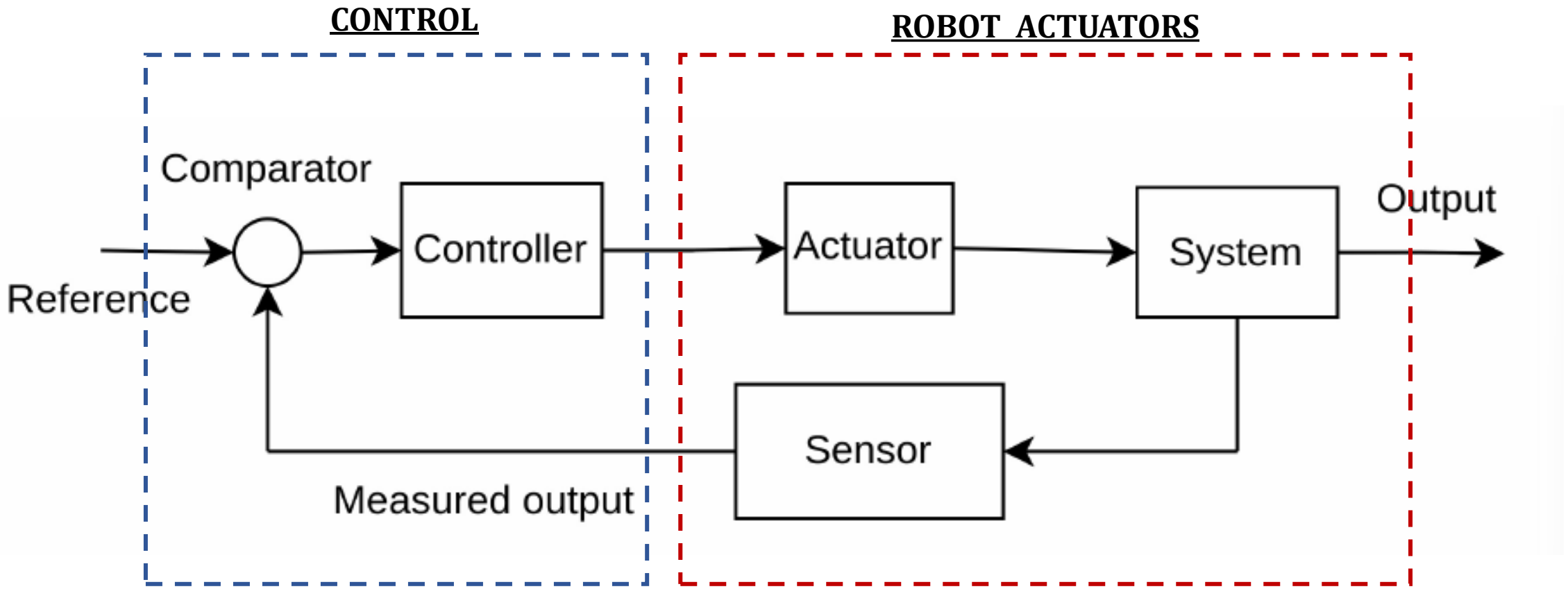
Layer	Purpose	Time Scale	Typical Input	Typical Output
Task Planning	Decides *what* to do: high-level goals and action sequences to achieve a mission.	Seconds → Minutes	Mission objective, world model, available actions	Sequence of actions or goals
Motion Planning	Decides *how* to move to reach each goal: 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



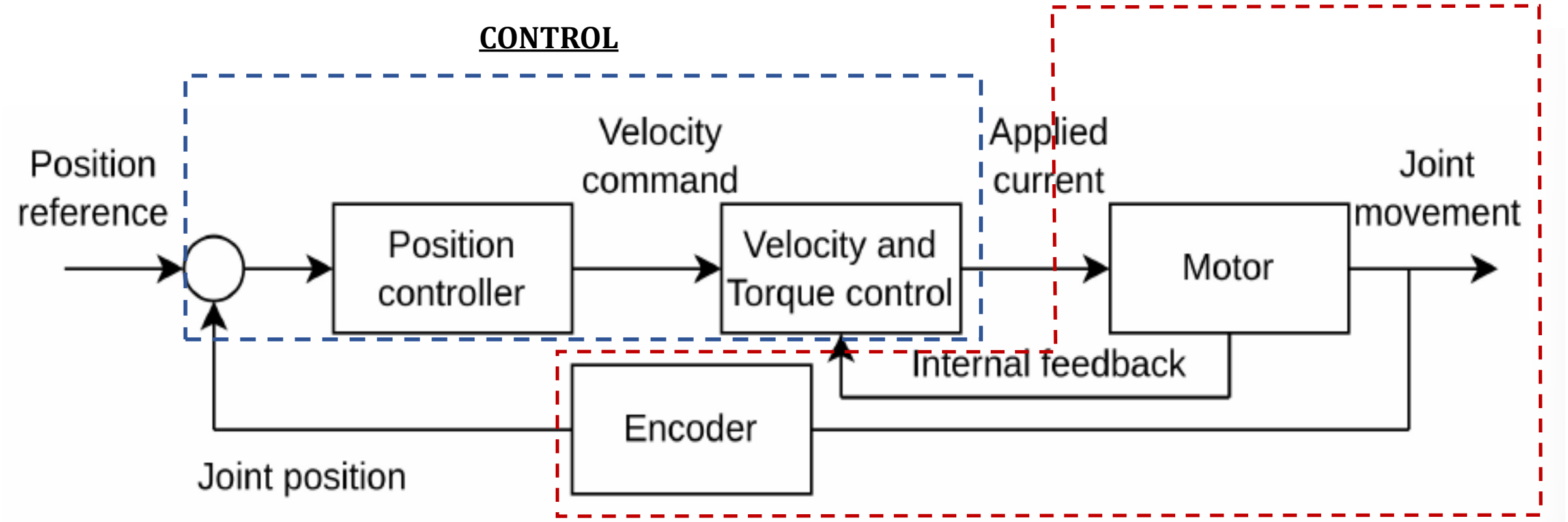
Closed Loop Control System



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:**

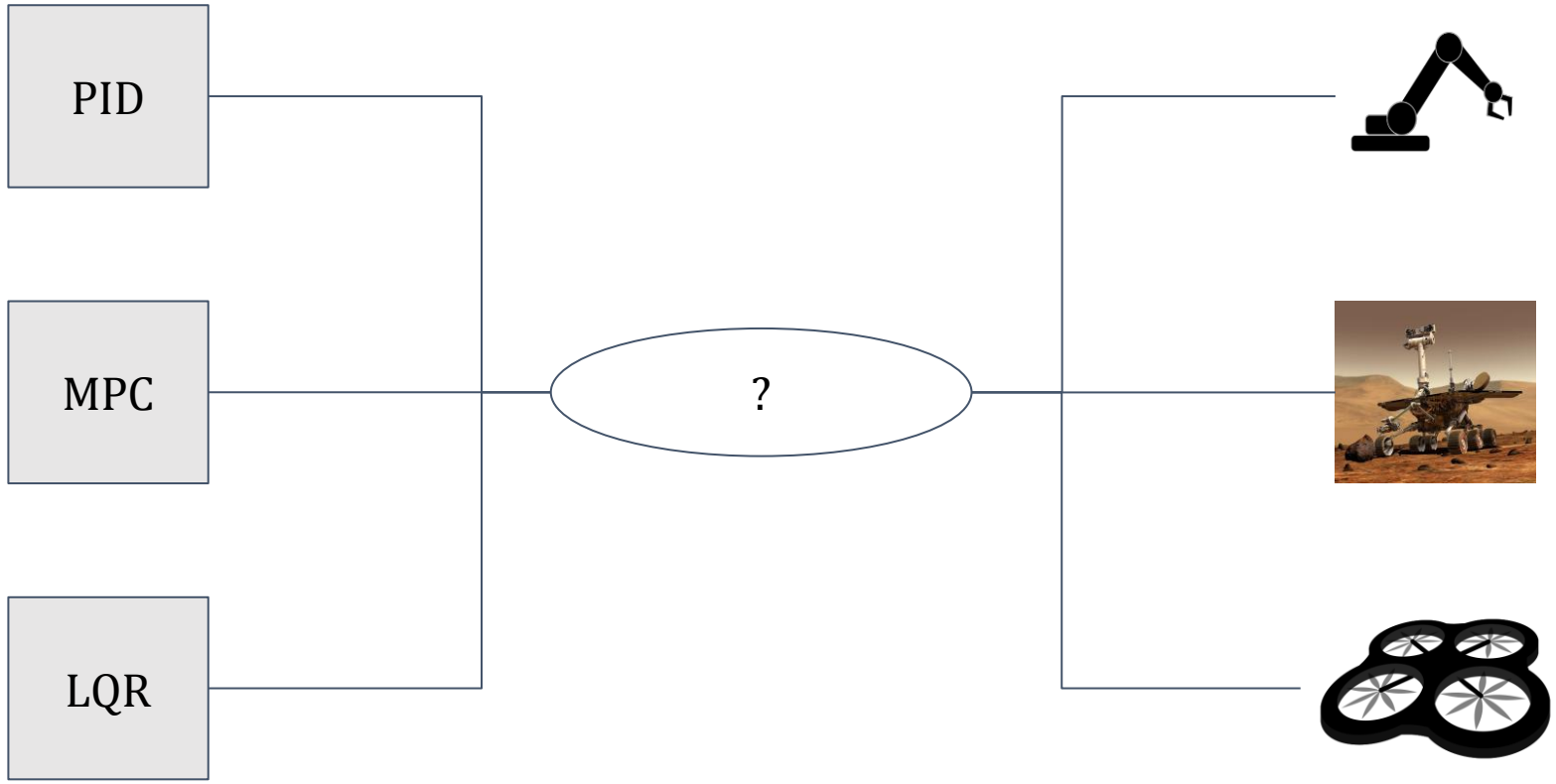
Use real-time-grade hardware	Choose actuators, sensors, and fieldbus systems designed for **deterministic communication** to minimize unpredictable delays.
Run on a real-time OS	Use a **real-time kernel** (e.g., PREEMPT_RT Linux) and configure your controller for **real-time scheduling**. Real-time ≠ 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

Control Algorithms

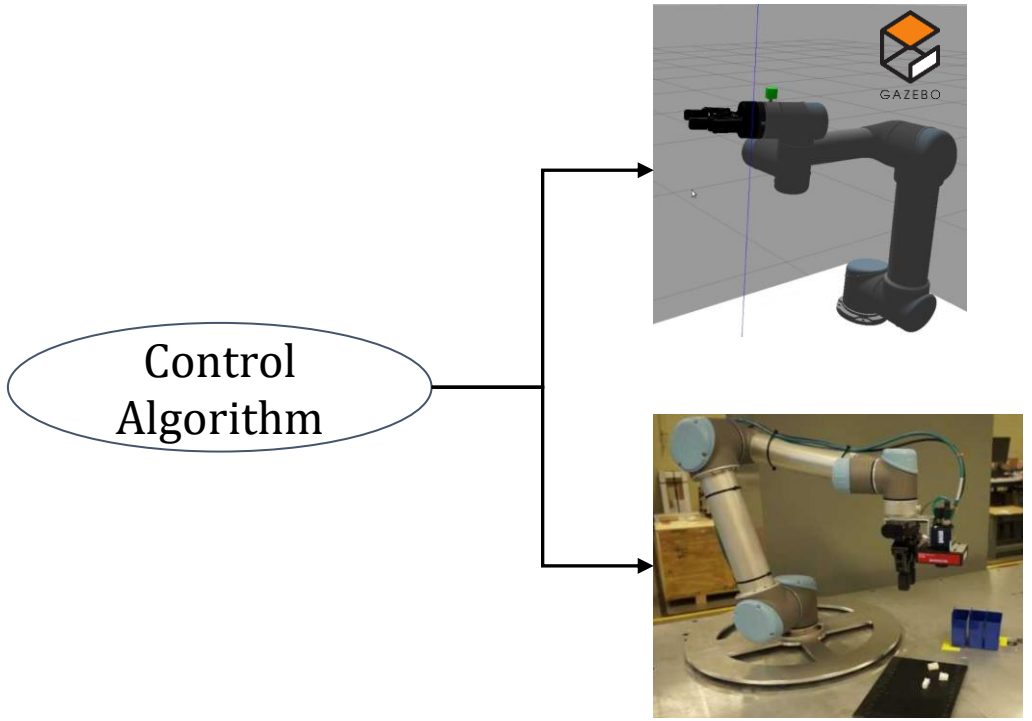
Hardware/Drivers



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**.



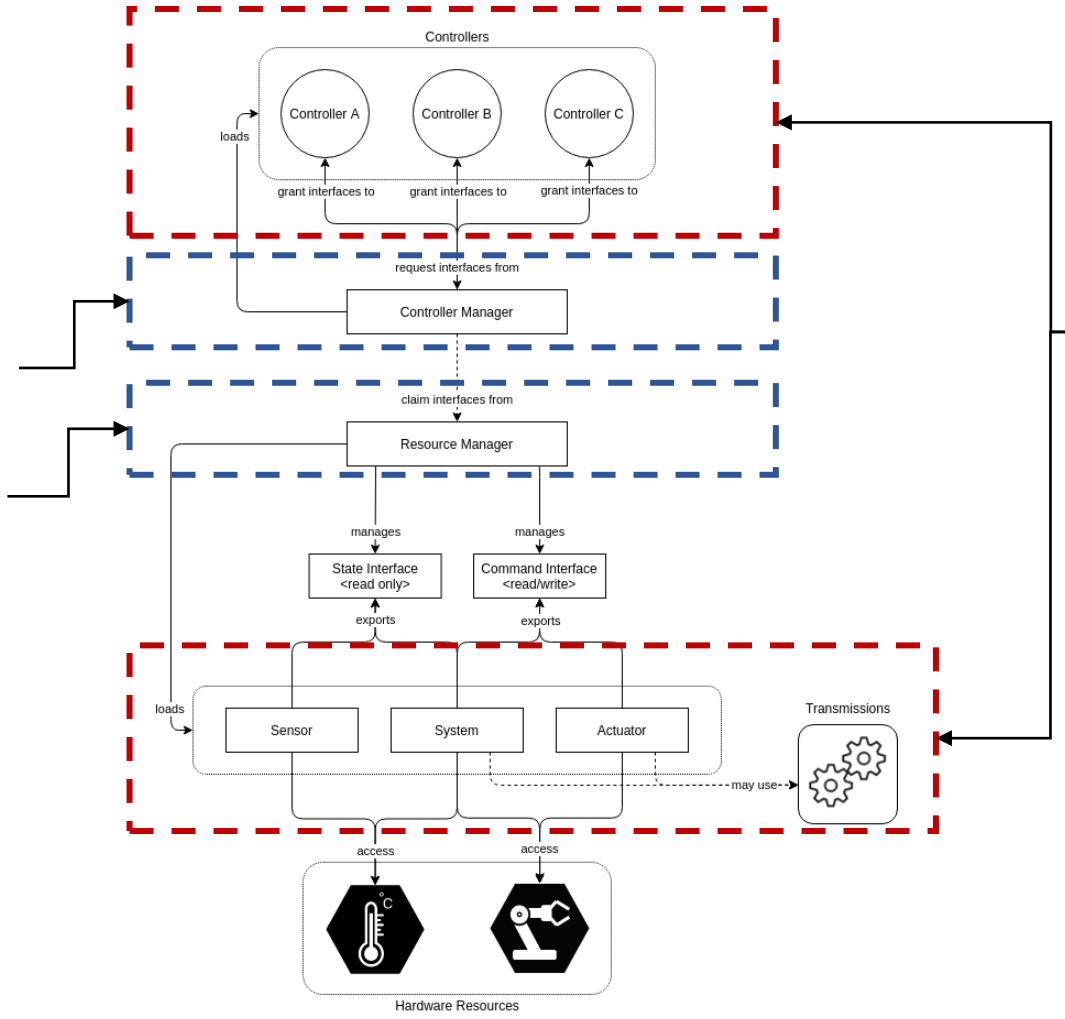
The framework ros2_control allows to do this.

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

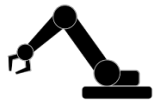
Plugins and Pluginlib

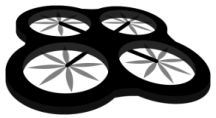
- Plugins are **dynamically loadable classes** that are loaded from a runtime library (i.e. shared object, dynamically linked library).
- Plugins are useful for **extending/modifying application behavior** without needing the application source code.

- *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.

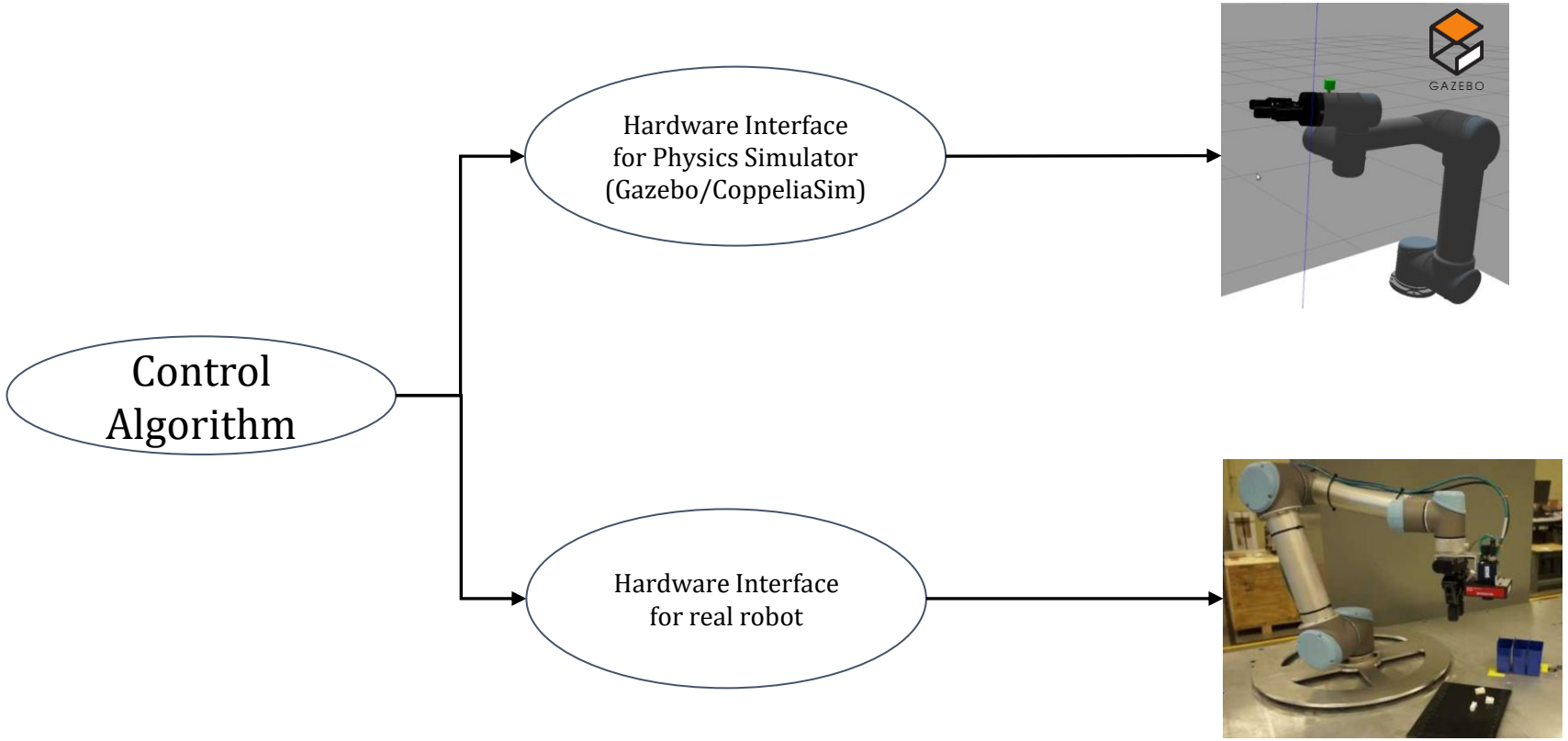
- Each category's behaviour is defined with a hardware **plugin**: `on_init()`, `on_configure()`, `read()`, `write()`.
- For each joint, the available command/state interfaces (position, velocity, effort) are declared.
- Examples of the URDF tags are available here.

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

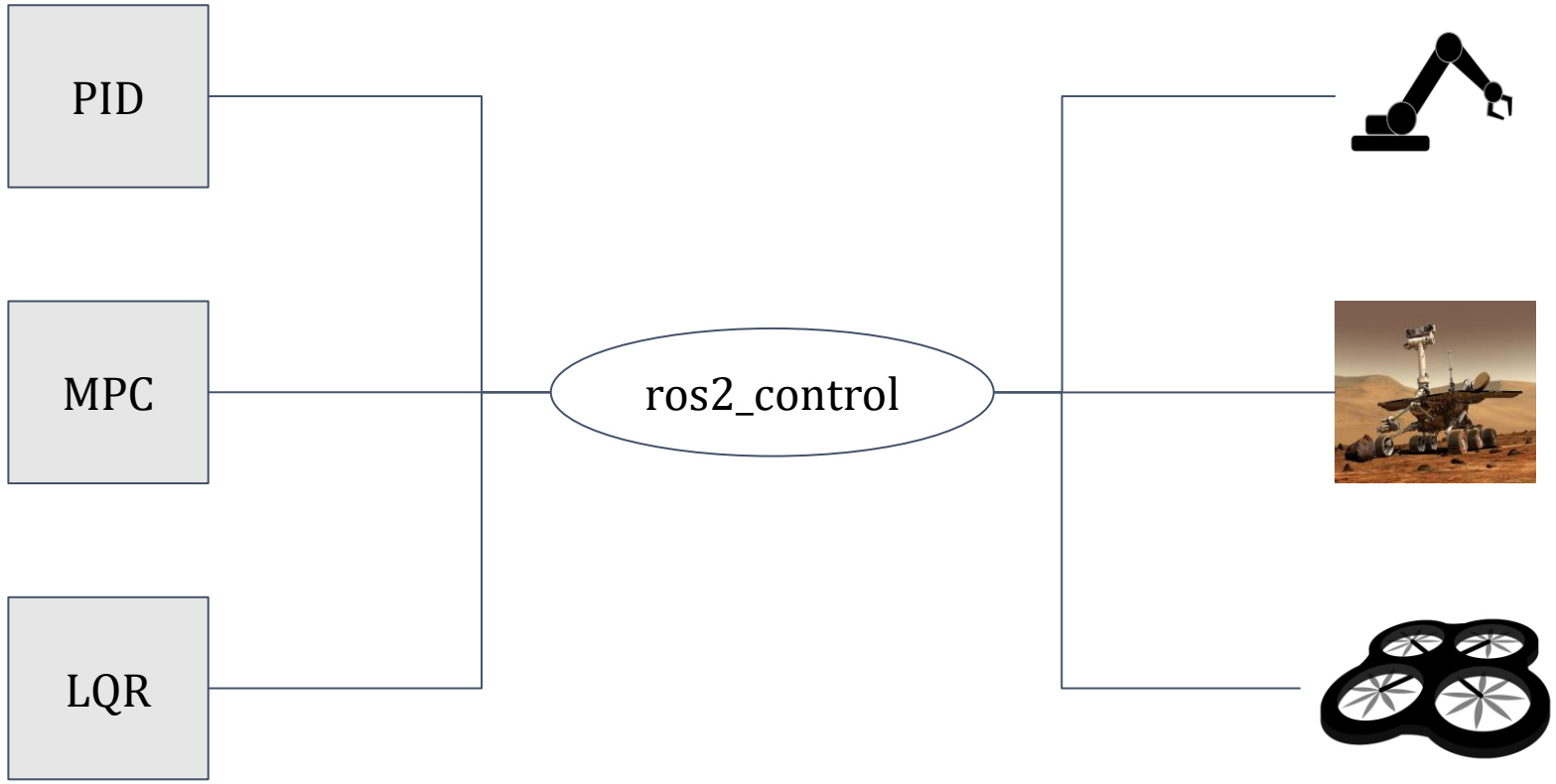
Only one hardware interface (Simulation/Real Hardware) needs to active at a time!



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).

UNIVERSITÀ
DI TRENTO
Department of
Information Engineering and Computer Science

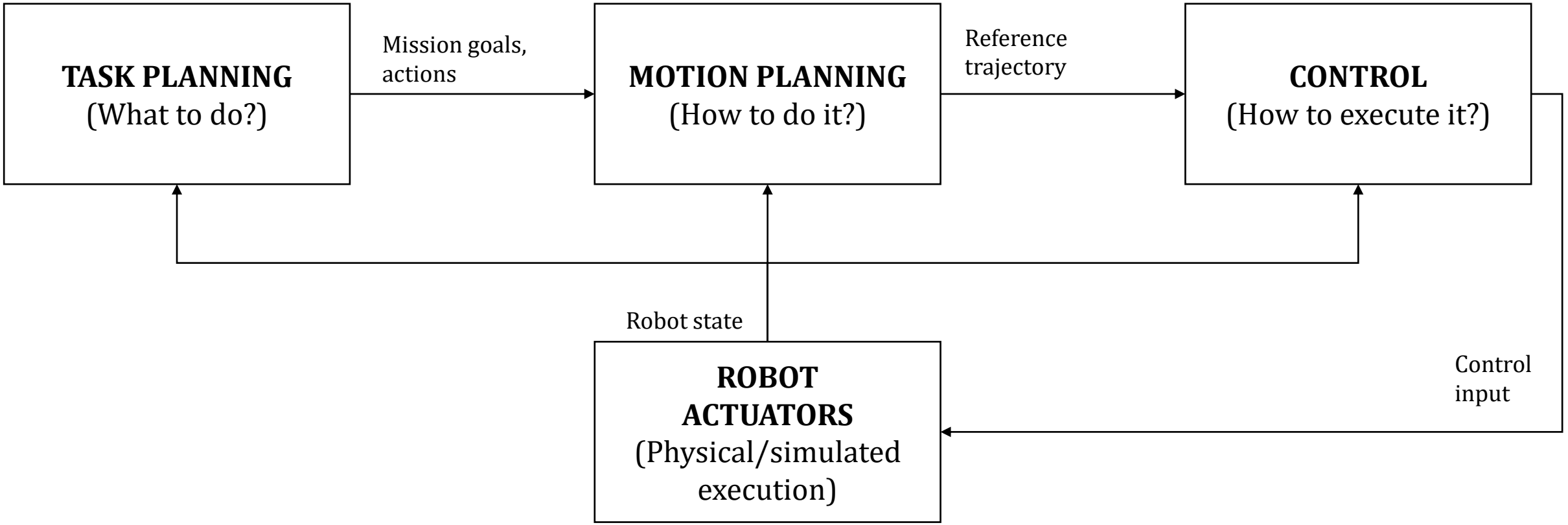
ROS 2 Motion and Task Planning

Edoardo Lamon

Software Development for Collaborative Robots

Academic Year 2025/26

High-level Framework for a Robotic Application

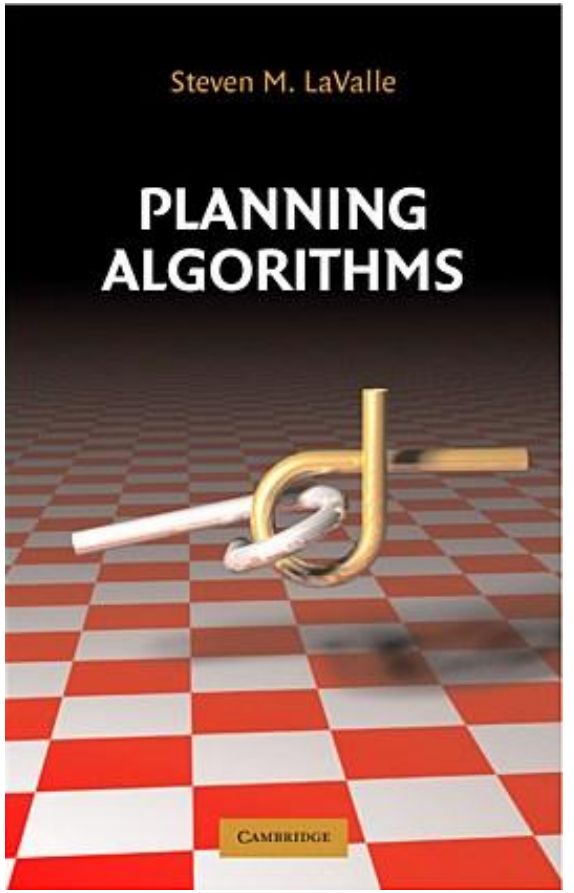


High-level Framework for a Robotic Application

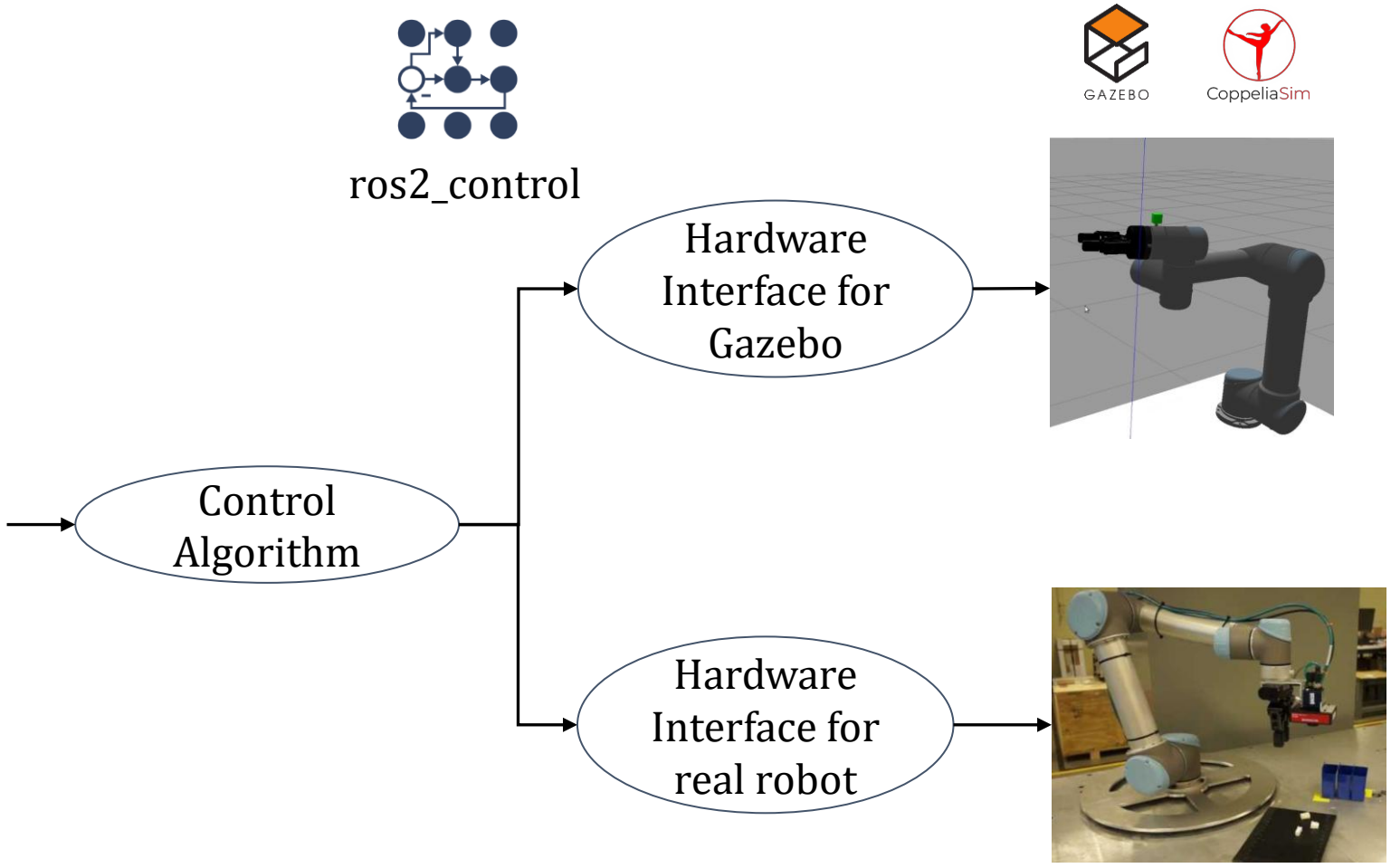
Layer	Purpose	Time Scale	Typical Input	Typical Output
Task Planning	Decides *what* to do: high-level goals and action sequences to achieve a mission.	Seconds → Minutes	Mission objective, world model, available actions	Sequence of actions or goals
Motion Planning	Decides *how* to move to reach each goal: 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

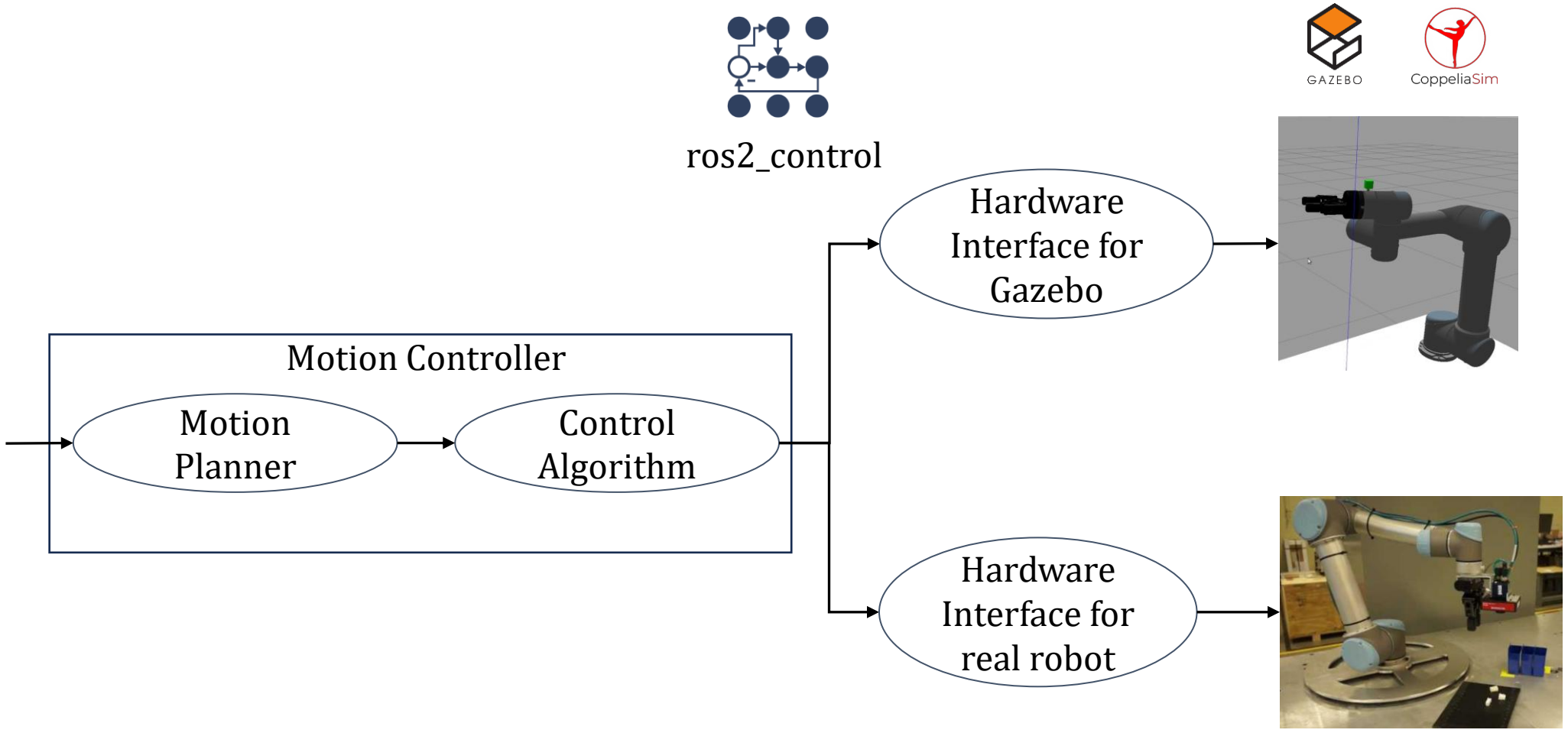
Motion Planners in ROS 2



Robot Programming Pipeline



Robot Programming Pipeline

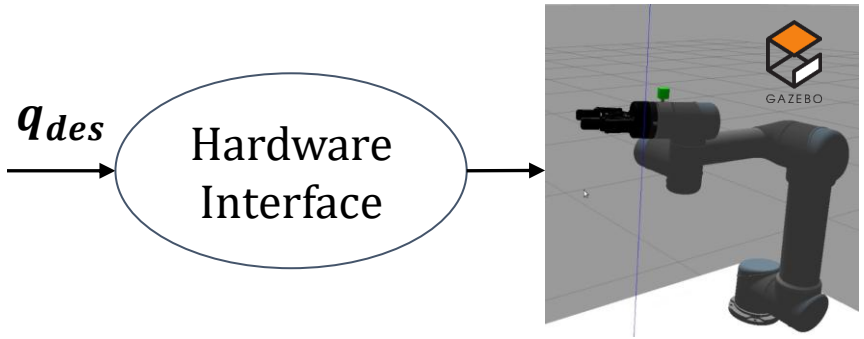


Motion Planning

- A control algorithm ensures that, given a predefined **reference** (either, position, velocity, acceleration, torque, force, etc.), the system is able to track the reference (low-level, usually in joint space and includes physical limits of the system);
- Each reference is assumed to be tracked in **short time** (one or few control loops, the reference is close to the actual value);
- Usually, the control references are generated by sampling higher level **trajectories (path+timings)** usually in Cartesian space (various control loops and depends on the distance of the actual position from the goal).
- Generally, within trajectories one can include **motion constraints** (self-collisions, obstacles etc.);
- Motion Planning and Control problems are really close and not always are handled separately. In general, one can refer to the Motion Control problem.

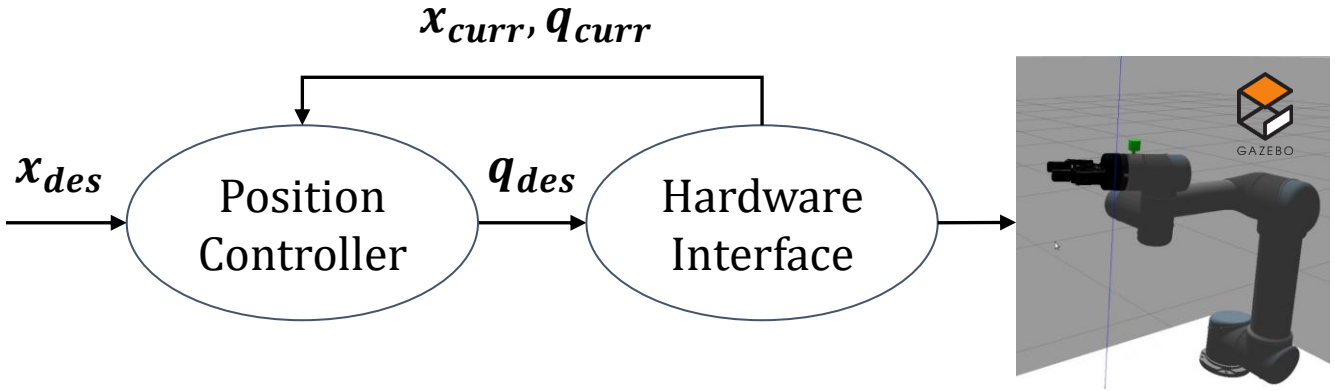
Motion Control example

Let's suppose to have an UR5 with a hardware interface which exposes joint position → We can set the target joint configuration $\boldsymbol{q_{des}}$.



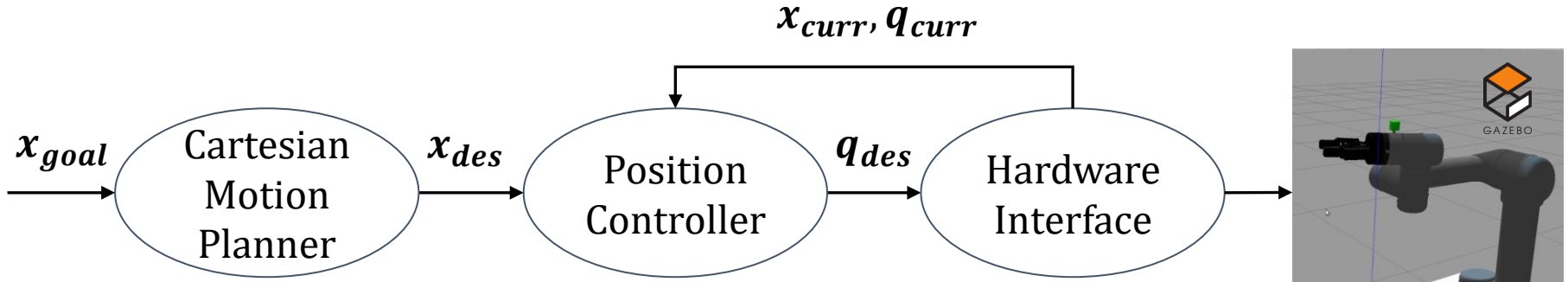
Motion Control example

A Cartesian position controller can receive a desired pose in the Cartesian space $\boldsymbol{x_{des}}$, read the current pose $\boldsymbol{x_{curr}}$ and configuration $\boldsymbol{q_{curr}}$ and map it to a desired configuration $\boldsymbol{q_{des}}$ (for example through inverse kinematics).



Motion Control example

A Cartesian motion planner can receive a goal pose $\boldsymbol{x_{goal}}$ which can be arbitrarily far from the current pose $\boldsymbol{x_{curr}}$, generate a trajectory with some patterns of motion (linear, curved, etc.), including also environmental constraints such as obstacles in the path. The trajectory is then sampled with step **dt** (control loop rate) to generate $\boldsymbol{x_{des}}$ and changes the position controller reference every **dt**.



OPEN CHAIN
(the current state is read only once, no feedback)

FEEDBACK LOOP
(every loop the current state is updated)

ROS2 Motion Planning

ROS 2 integrates two main frameworks for motion planning:

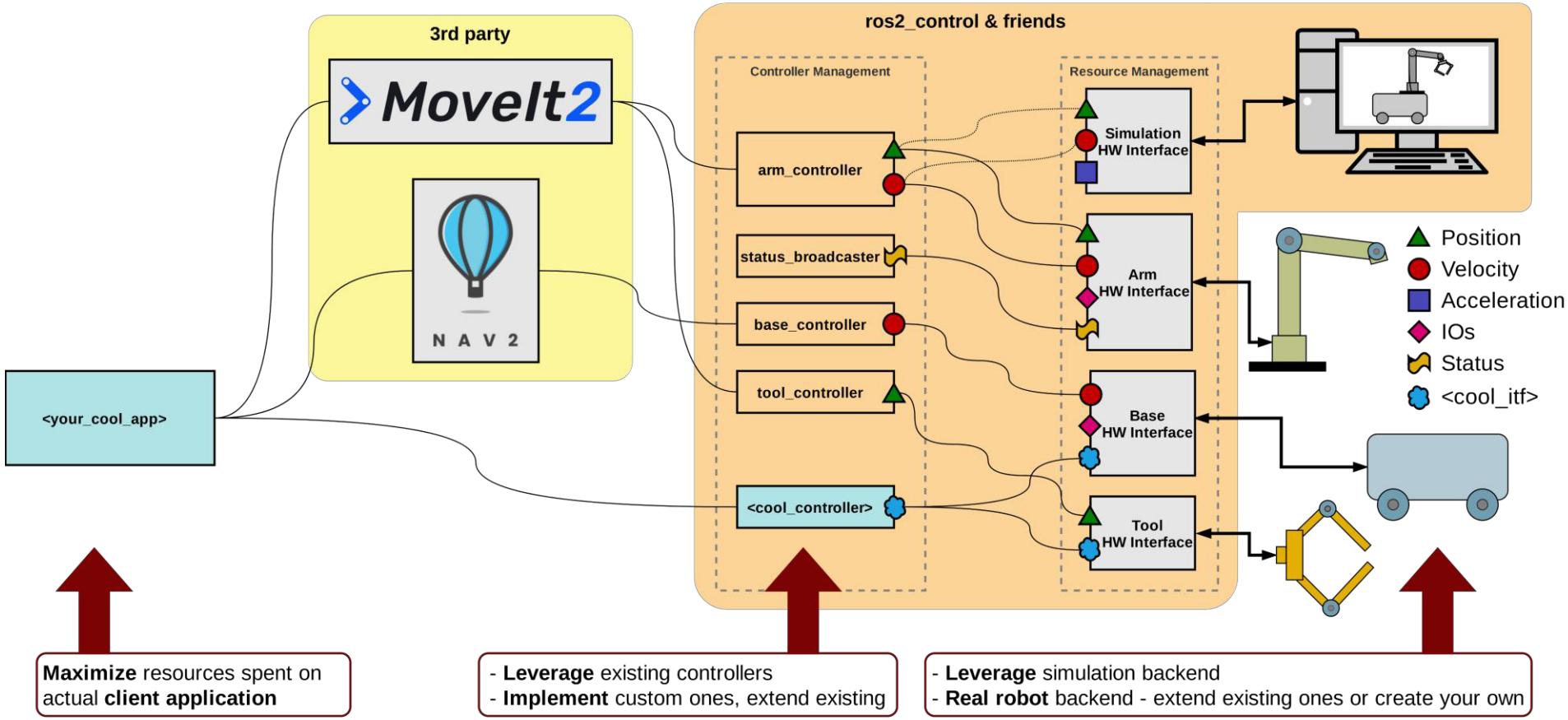
- MoveIt for manipulation (robotic arms);
- Nav2 for locomotion (mobile robots).

Frameworks are convenient, but often require long setup times and it is not always convenient to use them (depends on the features that one requires).

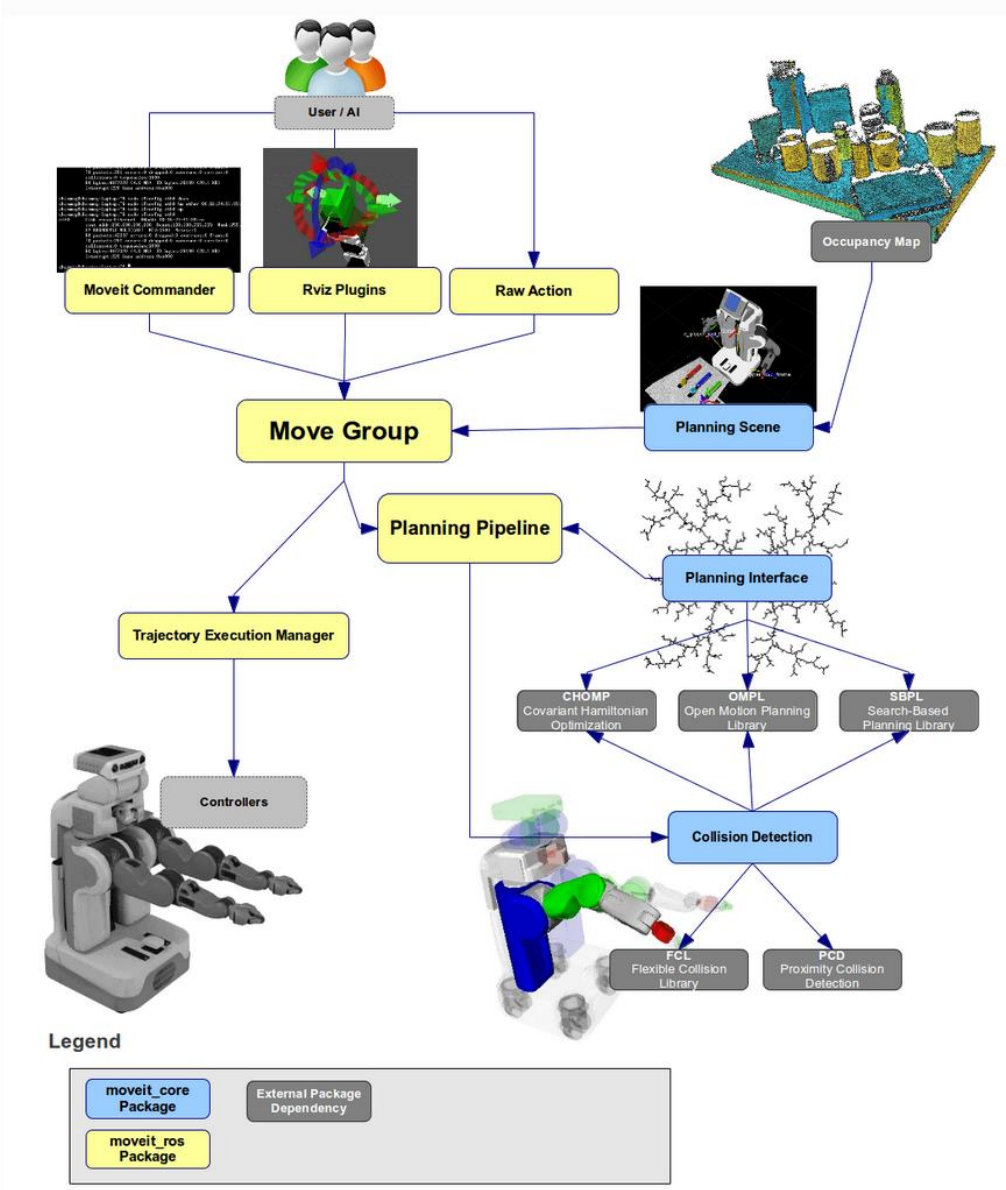
Othe options:

- Curobo from Nvidia;

ROS2 Motion Planning



MoveIt2


User / AI
Moveit Commander
Rviz Plugins
Raw Action
Occupancy Map
Planning Scene
Move Group
Planning Pipeline
Planning Interface
Trajectory Execution Manager
CHOMP
Covariant Hamiltonian Optimization
OMPL
Open Motion Planning Library
SBPL
Search-Based Planning Library
Controllers
Collision Detection
FCL
Flexible Collision Library
PCD
Proximity Collision Detection
Legend
moveit_core Package
External Package Dependency
moveit_ros Package

MoveIt2

MoveIt2 Component	Role / Function
Planning Scene	Represents the **environment** where the robot operates. Includes static geometry (meshes, polygons) and **dynamic obstacles** from sensors (lidar, cameras).
Planning Pipeline	Computes **collision-free trajectories** using robot kinematics and the planning scene. Supports **hybrid planning** combining global and local planners.
Move Group	The main **user interface node**. Receives planning requests, coordinates data collection, calls the planning pipeline, and sends the trajectory to execution. A robot may have multiple move groups (e.g., arm, gripper).
Trajectory Execution Manager	Sends planned trajectories to **controllers** via **ROS 2 actions**, monitors execution, and updates paths if the environment changes.
User Interfaces	- **RViz plugin:** visualize and interact with the robot and generated trajectories. - **CLI tools:** send planning or execution commands from the terminal.

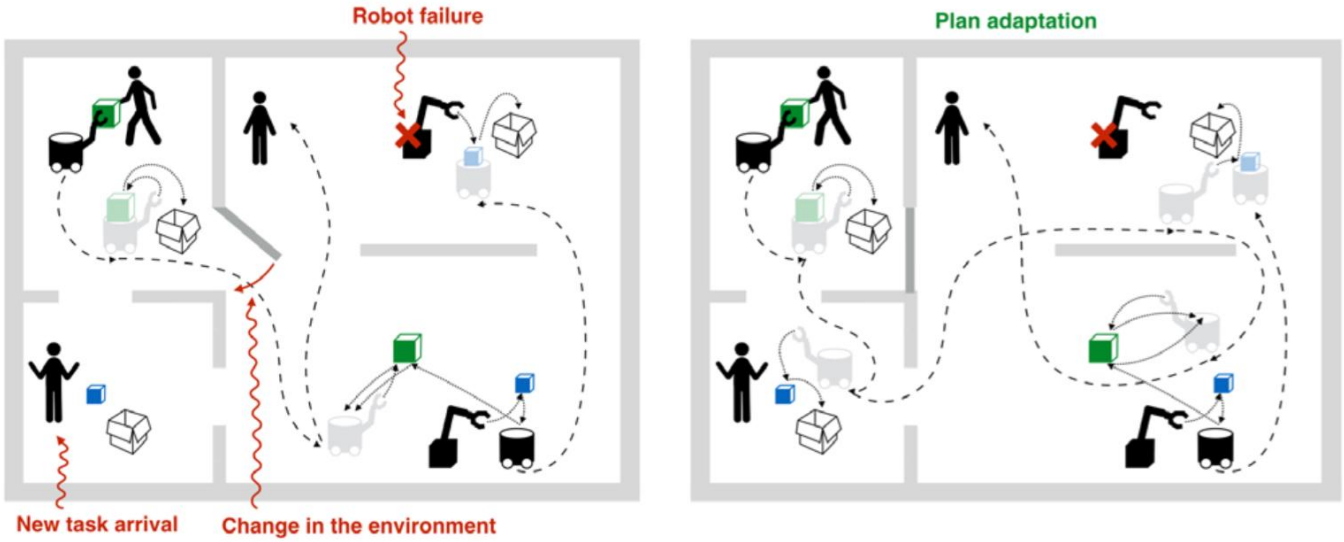
MoveIt2

MoveIt provides ROS 2 packages to plan manipulation tasks and integrates several libraries:

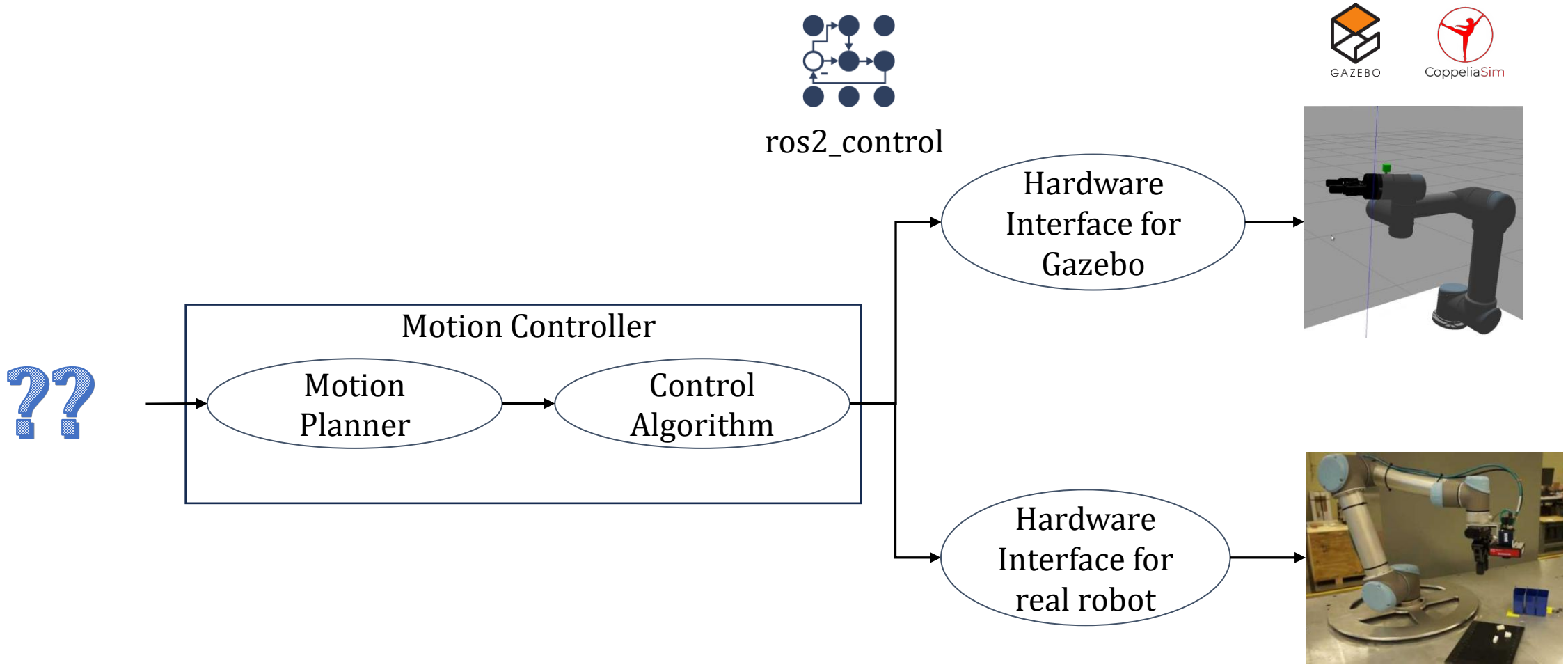
- The Kinematics and Dynamics Library (KDL) for modelling and computation of kinematic chains.
- Inverse Kinematics solvers like IKfast from OPENRAVE, or TrakIK;
- The Open Motion Planning Library (OMPL), a collection of state-of-the-art sampling-based motion planning algorithms.
- The Stochastic Trajectory Optimization for Motion Planning (STOMP), a probabilistic optimization framework for collision-free paths;
- The Trajectory Optimization Motion Planner (TrajOpt), a sequential convex optimization algorithm for motion planning problems.
- And others (SRMP ...)

ROS 2 Tools

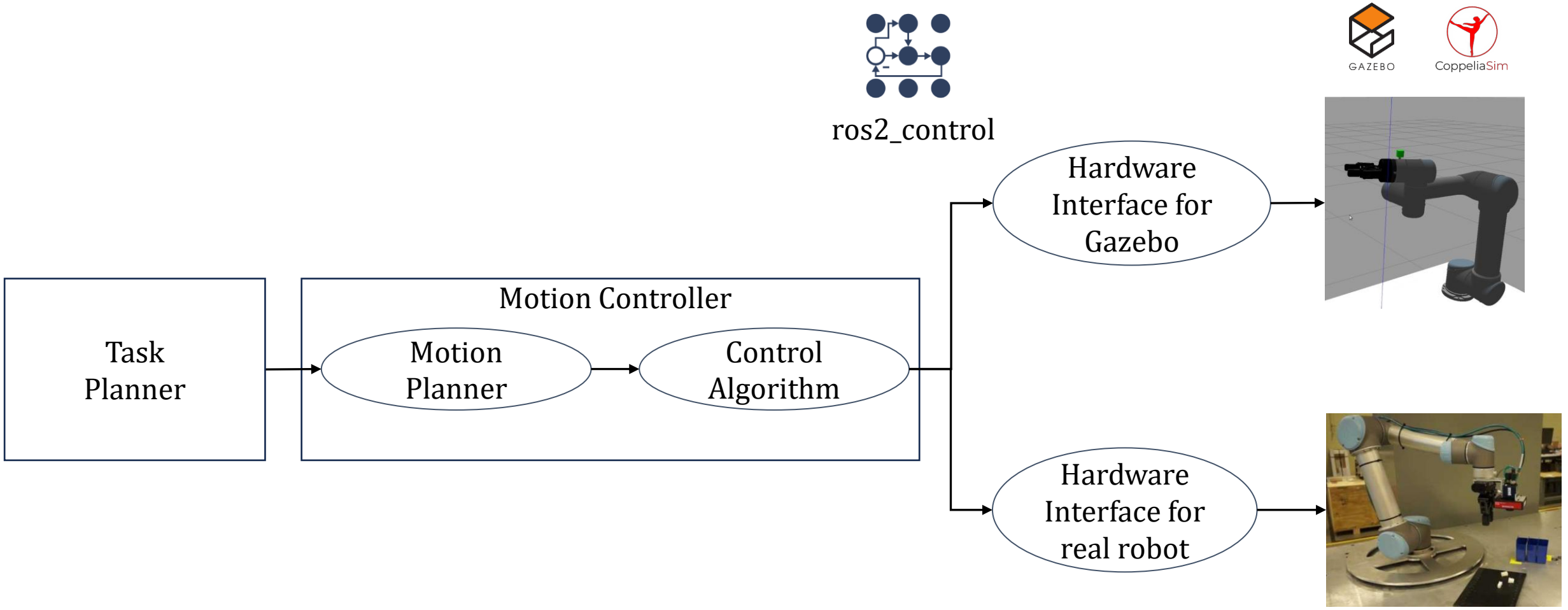
Task Planners in ROS 2



Robot Programming Pipeline



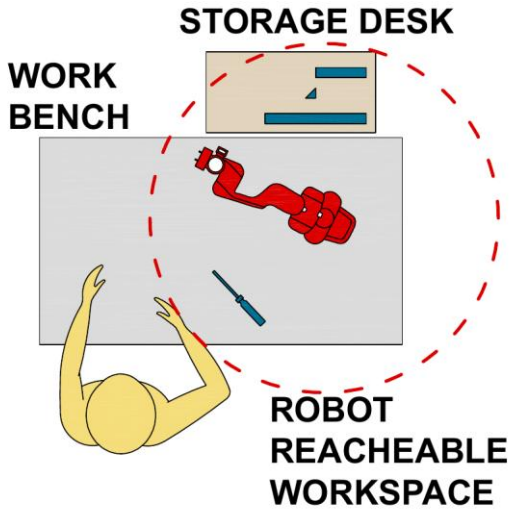
Robot Programming Pipeline



Task Planning example

- Suppose we have a manipulator in a world like the one in the picture. We would like our robot to perform a task such as *"take the objects from the storage desk and put it on the work bench"*.
- We could describe what the robot should do by breaking the solution down into individual *actions:*

  1. **Move** to the *desk*
  2. **Pick** up the *object*
  3. **Move** to the *work bench*
  4. **Place** the *object*



Task Planning example

- What if the task gets more complicated? Suppose we add more objects and locations to our simple world, and start describing increasingly complex goals like *“make sure all the screws are on the work bench and everything else is in the desk”*?
- Furthermore, what if we want to achieve this in some optimal manner like minimizing time, or number of total actions? Our reasoning quickly hits its limit as the problem grows, and perhaps we want to consider letting a computer do the work.

Task Planning

- Task planning refers to the autonomous reasoning about the state of the world using an internal *model* and coming up a sequence of *actions*, or a *plan*, to achieve a *goal*.

- It requires a model of the world and how an autonomous agent can interact with the world. This is usually described using *state* and *actions*. Given a particular state of the world, our robot can take some action to *transition* to another state.

- For example, in Finite State Machine (FSM), the goal is formulated as the final state.

Task Planning

- Task planning refers to the autonomous reasoning about the state of the world using an internal *model* and coming up a sequence of *actions*, or a *plan*, to achieve a *goal*.

- It requires a model of the world and how an autonomous agent can interact with the world. This is usually described using *state* and *actions*. Given a particular state of the world, our robot can take some action to *transition* to another state.

- For example, in Finite State Machine (FSM), the goal is formulated as the final state.

Task Planning features

- **Easy introspection**: figuring out what the robot is going to do given its state and inputs shouldn't be too complicated.
- **Maintainability**: the system should be easily modifiable.
- **Resilience**: the system should be capable of dealing with unexpected circumstances in the environment.
- **Modularity** and **scalability**: the components, and those actions composed by the interaction of multiple components, should be reusable not only within one system, but also in heterogeneous systems (for example in different robots).
- **Efficiency**: the organization should happen without significant computational overhead.
- **Separation of concerns**: each component behavior depends only on a limited amount of inputs → the logic controlling the execution flow of the machine needs to be separated from the actions implementation.

Task Planning in robotics

Task planning is an AI problem and robotics is only one of its main application field. Thus:

- Best task planners are agent and task agnostic;
- Actions are a more general class of interactions which include motions;
- Some task planners require a Knowledge Base (e.g., PDDL, Prolog, etc.);
- Most of task planners are not integrated with ROS2.

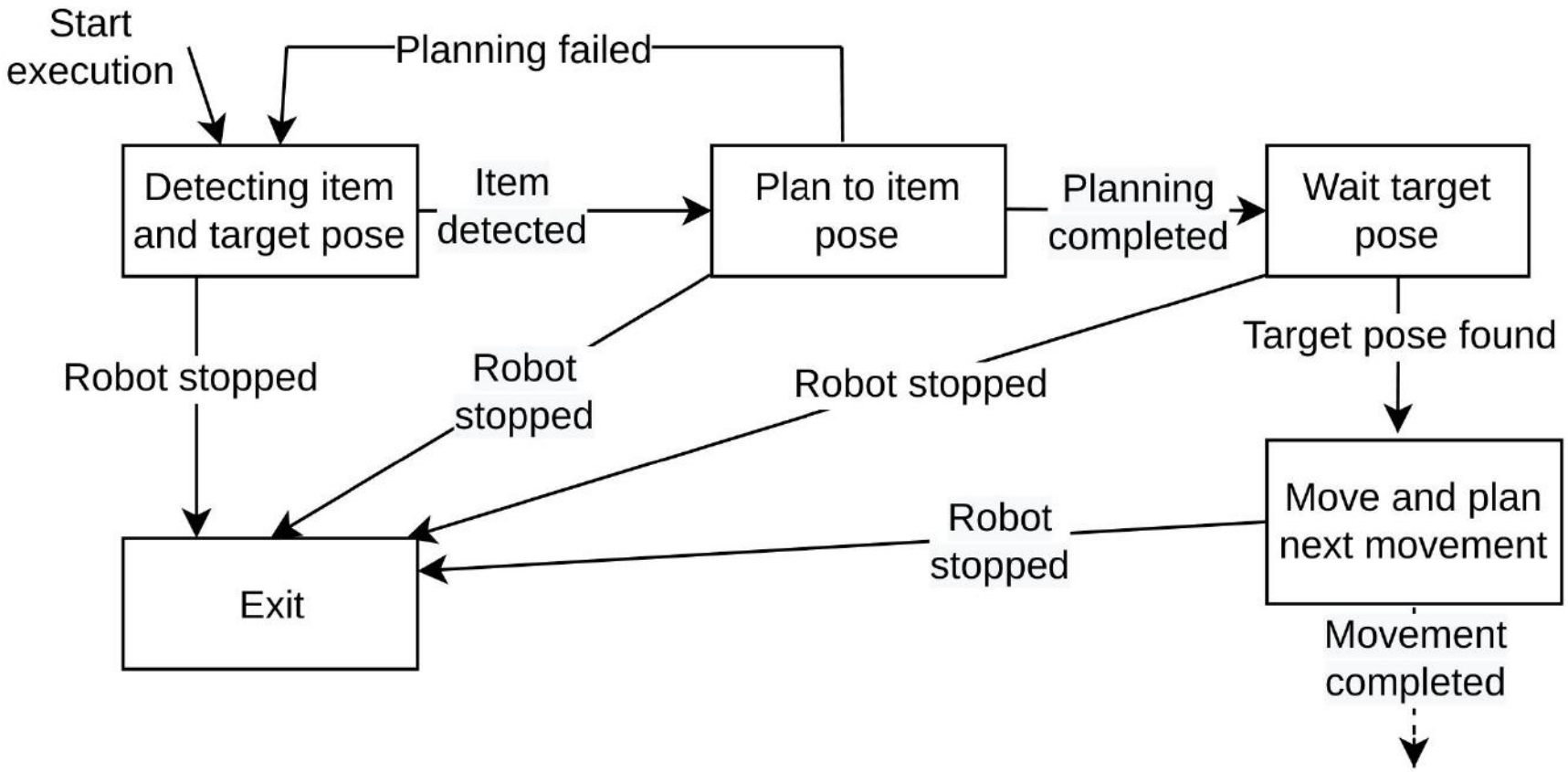
Finite State Machines (FSMs)

A **Finite State Machine (FSM)** models a system as a **finite set of states** and **rules for transitioning** between them.

At any time:

- The system is in **one and only one state**.
- Transitions occur **only if defined conditions are met**.
- **Different actions** are executed depending on the current state or transition.

Finite State Machines (FSMs)



Finite State Machines (FSMs)

Issue	Description
Explosion of transitions	As the number of states grows, the number of possible transitions increases even faster, especially for error handling.
Hard to visualize	Large FSMs become complex to represent and maintain, losing their initial readability.
Limited reusability	Transitions and logic are often tied to local variables, making code reuse difficult.

Solution: **Hierarchical Finite State Machines (HFSMs)**

- Nested states: States can contain **sub-states**, forming modular sub–state machines.
- Grouped logic: Related behaviors are **encapsulated** inside macro-states (e.g., *Manipulation* contains *Plan*, *Move*, *Grasp*).
- Simpler transitions: Transitions can occur between sub-states and higher-level states, reducing redundancy
- More modular: Encourages reuse and easier maintenance for large robotic systems.

Behavior trees

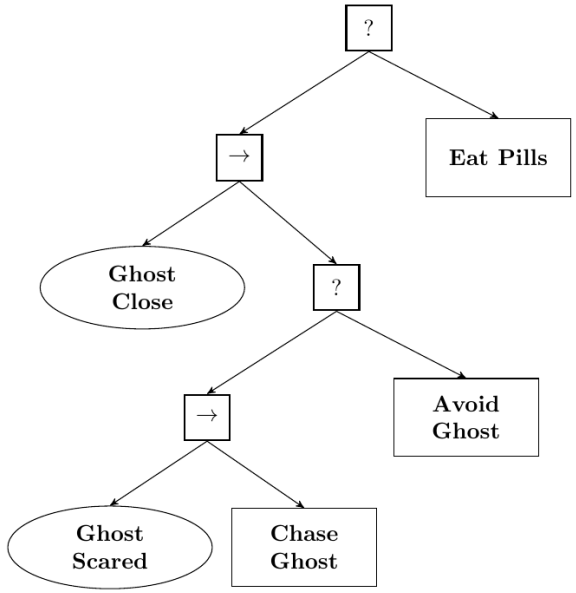
A **Behavior Tree** organizes tasks and conditions into a **hierarchical tree** traversed from **root → leaves** at each *tick*. Each **leaf node** represents an **action** or **condition**, while **control nodes** decide the execution flow.

Execution:

- The tree is **"ticked" periodically** or when an **event** occurs.
- The tick travels **from root to leaves**, returning *success*, *failure*, or *running*.
- Shared data between nodes is stored in a **blackboard**.

Behavior trees

Type of Node	Symbol	Success	Failure	Running
Sequence	$\rightarrow$	All children succeed	One child fails	One child returns Running
Fallback	?	One child succeeds	All children fail	One child returns Running
Decorator	$\diamond$	Custom	Custom	Custom
Parallel	$\rightrightarrows$	$\geq M$ children succeed	$> N - M$ children fail	else
Condition	$\bigcirc$	True	False	Never
Action	$\square$	Upon completion	Impossible to complete	During execution



Behavior trees

Reactivity

- Each task or condition quickly returns: ✅ *Success*, ❌ *Failure*, or ⏳ *Running*
- Enables **fast response** to environment or system changes
- Long-running tasks execute **asynchronously**, keeping the BT responsive

Modularity

- Branches can be **reused as subtrees**, promoting **hierarchical organization**
- **Add/remove behaviors** without redefining transitions
- Encourages **clean, maintainable, and scalable** designs

Readability

- **Graphical representation** makes robot logic easier to visualize and debug
- Once familiar with BT symbols, users can **interpret and modify** behavior intuitively
- Combines structure and clarity, even for complex systems

ROS2 Task Planning Libraries

Integration with ROS2:

- SMACC2 State Machine library, an event-driven, asynchronous, behavioral state machine library for real-time ROS 2 applications written in C++.
- Yasmin, implements robot behaviors using Finite State Machines (FSM). It is available for ROS 2, Python and C++.
- Behavior Trees, a tree-based planning system based on actions compositions (not state transitions). Both C++ implementation and ROS2 integration are available.

For generating autonomous behaviors without the need of a static program?

- FlexBE model behaviors as hierarchical state machines where states correspond to active actions and transitions describe the reaction to outcomes;
- Skiros2 implements an extension of the Behavior trees with parameters and conditions to form primitive and compound skills;
- PlanSys2 (inspired by ROSPlan), a PDDL(Planning Domain Definition Language)-based planning system implemented in ROS2.

UNIVERSITÀ
DI TRENTO
Department of
Information Engineering and Computer Science

Testing Your Code

Edoardo Lamon

Software Development for Collaborative Robots

Academic Year 2025/26

Why “pure software” testing?

In the development of a robotic application, the testing on the robot hardware is the fun part (that many AI/CV apps do not have). However, this is also a delicate process:

- Hardware is **expensive** (mistakes can cause **damage**);
- Limited **availability** (maybe it is shared with other developers);
- **Repeatability** can be difficult to achieve in real-world conditions;
- Debugging the software in case of a hardware test failure can be **time-consuming and challenging,** making it difficult to reproduce errors.

Before going to the hardware, we want to detect and remove the software bugs.

Test Driven Development (TDD)

TDD is a **software development paradigm** where you **write the tests before writing the code.**

The TDD cycle:

1. **Write a test** based on system requirements.
2. **Run it → it fails** (no code yet!).
3. **Implement** the minimal code needed to pass the test.
4. **Run all tests again;** iterate until everything passes.
5. **Refactor** and improve the code while keeping tests green.

Why It Works?

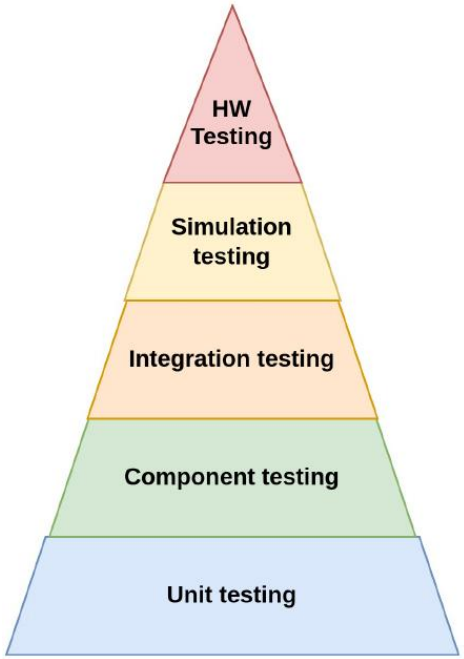
- Forces **clear requirements** before coding.
- Encourages **testable, modular design** from the start.
- Prevents "retrofitting" tests to poorly structured code.
- Makes debugging faster and safer.

When your system requirements are well-defined, **start from the tests,** not the code!

Testing in ROS2

The modularity of ROS2 helps a lot:

- ROS 2 applications are composed of many **independent building blocks** (packages, nodes, libraries).
- Each block can be **tested in isolation**, ensuring it behaves as expected before integration.
- This **reduces complexity** and increases trust in higher-level system tests.



Test Level	Scope	Goal
Unit Tests	Individual functions or libraries	Verify correctness of small, isolated pieces of code
Node Tests	Single ROS 2 nodes	Ensure correct ROS API behavior (topics, services, actions)
Integration Tests	Groups of nodes	Validate communication and interaction between components
System Tests	Full application	Verify end-to-end functionality in a realistic environment

Unit Testing

Verify that a small piece of code (function, class, or method) produces the expected output for a given input.

Advantages

- **Easier debugging** → failures point directly to the responsible function.
- **Improved readability** → tests act as **"executable documentation"**, helping understand legacy code.
- **Fine control** → allows targeted validation of low-level logic.

Disadvantages

- Very fine-grained testing can cause **maintenance overhead**: small code changes may require updating many tests.
- Find a **balance:** more detailed tests for **stable core functions**, broader ones for **frequently changing code**.

Unit Testing in ROS2

Unit test focus on isolated components, **without involving ROS communication.**

→We can use standard tools for unit testing without any adaptation;

→the ROS2 tools will simply make it easier for you to install and execute them with colcon test

Language	**Testing Framework**	**Integration in ROS 2**
C++	**GTest** (Google Test)	via `ament_add_gtest()`
Python	**Pytest**/**unittest**	via `ament_add_pytest_test()`

GTest in ROS 2

Google Test (GTest) is a popular, open-source framework for **C/C++ unit testing.**

- Common default choice for **ROS 2 C++ projects**, especially those interacting with **hardware.**
- Supports **mocking** (simulated interfaces) — perfect for testing code without touching real devices.

To define one or more tests, Gtests provides a **set of macros** that are automatically discovered at initialization time. *Tests* use assertions to verify the tested code's behavior. If a test crashes or has a failed assertion, then it *fails*; otherwise, it *succeeds*.

```
1 TEST(TestSuiteName, TestName) {
2   ... Run some stuff
3   ... Make some asserions
4 }
```

The **assertions** are another set of macros that allow you to verify that the outputs of the algorithm you are testing are within the expected ones. An assertion's result can be *success*, *nonfatal failure*, or *fatal failure*. If a fatal failure occurs, it aborts the current function; otherwise, the program continues normally.

GTest in ROS 2

```
#include <gtest/gtest.h>
#include "rclcpp/rclcpp.hpp"

TEST(MyNodeTest, simple_check)
{
  auto node = std::make_shared<rclcpp::Node>("test_node");
  ASSERT_TRUE(node->count_publishers("/chatter") == 1);
}

int main(int argc, char **argv)
{
  testing::InitGoogleTest(&argc, argv);
  rclcpp::init(argc, argv);
  int result = RUN_ALL_TESTS();
  rclcpp::shutdown();
  return result;
}
```

CMake Integration:

```
ament_add_gtest(my_test test/my_test.cpp)
ament_target_dependencies(my_test rclcpp)
```

Then simply run:

```
colcon test
```

GTest in ROS 2 – Text Fixtures

- When several tests share **setup or common data**, use a **fixture class** to avoid code repetition.
- The class derives from testing::Test.
- Each test **TEST_F** runs as a **method** of that class, with access to its members.
- The fixture *TestFixturesName* is created before each test (calling *SetUp()*) and destroyed after (calling *TearDown()*), ensuring isolation and clean state.

```
1
2  #include <rclcpp/rclcpp.hpp>
3  #include <gtest/gtest.h>
4
5  #include "a_class_using_ros.hpp"
6
7  class TestFixtureName : public testing::Test {
8   public:
9    TestFixtureName()
10       : node_(std::make_shared<rclcpp::Node>("
     test_with_node")){}
11
12   void SetUp() override {
13     // Any code that should execute before the test starts
14     node_->declare_parameter("param_name", std::string{"
     param_default_value"});
15   }
16
17   void TearDown() override {
18     // Any code that should execute after the test run
19   }
20
21  protected:
22   rclcpp::Node::SharedPtr node_;
23 };
24
25 TEST_F(TestFixtureName, TestMethodTrue) {
26   AClassUsingRos class = AClassUsingRos(node_);
27   EXPECT_TRUE(class.aClassMethodThatShouldReturnTrue());
28 }
29
30 TEST_F(TestFixtureName, AnotherTestName) {
31   EXPECT_EQ(aFunctionThatShouldReturnOne(node_), 1);
32 }
```

GTest in ROS 2 – Parametrized Text

- Use when the **same logic** must run with **different inputs**.
- TEST_P macro allows you to define a vector with different parameters against which your test will be executed.
- Define test parameters with INSTANTIATE_TEST_CASE_P macro.
- Access them via GetParam() inside the test.
- Multiple parameters can be passed with Tuples.

```
1
2  class ParamTestFixtureName : public ::testing::
       TestWithParam<std::tuple<int, int, bool>> { ... }
3
4  TEST_P(ParamTestFixtureName, TestSmaller) {
5      auto first_element = std::get<0>(GetParam());
6      auto second_element = std::get<1>(GetParam());
7      auto expected_result = std::get<2>(GetParam());
8      EXPECT_EQ(IsSmaller(first_element, second_element),
       expected_result);
9  }
10
11 INSTANTIATE_TEST_CASE_P(
12         BoringSmallerTests,
13         ParamTestFixtureName,
14         ::testing::Values(
15                 std::make_tuple(0, 1, true),
16                 std::make_tuple(1, 0, false),
17                 std::make_tuple(-5, -4, true));
```

GTest in ROS 2 – Mocks

- Robot software often needs to **talk to hardware** (motors, sensors, I/O boards).
- Plugging real devices in every time you test is **inconvenient** and **risky**.

→**Abstract** the hardware layer from your logic: **Dependency Injection Pattern** (code depends on **interfaces**, not concrete hardware classes.)

Approach	Description	Pros / Cons
Emulation	Build fake devices that simulate real behavior (using simulators!).	+ Realistic, - Costly to create/maintain
Mocking (via GMock)	Create mock classes that expect specific function calls.	+ Lightweight, + Quick to set up

GTest in ROS 2 – Mocks example

1. Start from the interface class:

Define an integrated umbrella used by the robot to protect itself from the rain.

```
1
2  class UmbrellaInterface {
3    virtual ~UmbrellaInterface() {};
4    virtual void openUmbrella() = 0;
5    virtual void closeUmbrella() = 0;
6    virtual bool isOpen() const = 0;
7  };
```

2. Pick a function that uses the interface:

It will open the umbrella when it's raining and close it when it's not.

```
1
2  class Umbrella : public UmbrellaInterface {
3    public:
4    void openUmbrella() override {
5      ... Advanced science stuff that interacts with real
       umbrellas
6    }
7    void closeUmbrella() override { ... }
8    bool isOpen() const oeverride { ... }
9  };
10
11 void manageUmbrella(bool its_raining, std::shared_ptr<
       UmbrellaInterface> umbrella) {
12   if(its_raining && !umbrella->isOpen())
13     umbrella->openUmbrella();
14   if(!its_raining && umbrella->isOpen())
15     umbrella->closeUmbrella();
16 }
```

GTest in ROS 2 – Mocks example

3. Create a mock of the Umbrella class

Fill in the macro MOCK_METHOD for each function using the signature "MOCK_METHOD(ReturnType, MethodName,(Args))"

```
1
2 #include "gmock/gmock.h"
3
4 class MockUmbrella : public UmbrellaInterface {
5  public:
6   MOCK_METHOD(void, openUmbrella, (), (override));
7   MOCK_METHOD(void, closeUmbrella, (), (override));
8   MOCK_METHOD(bool, isOpen, (), (const, override));
9 };
```

4. Write a test for the function:

The test verifies that, if it rains and the umbrella is not open, the function openUmbrella is called.

```
1
2  #include "mock_umbrella.hpp"
3  #include "headers/with/functions/we/want/to/test.hpp"
4  #include "gmock/gmock.h"
5  #include "gtest/gtest.h"
6
7  using ::testing::AtLeast;
8  using ::testing::Return;
9
10 TEST(UmbrellaTest, willOpenUmbrella) {
11   auto umbrella = std::make_shared<MockUmbrella>();
12   EXPECT_CALL(*umbrella, isOpen())
13       .WillOnce(Return(false));
14   EXPECT_CALL(*umbrella, openUmbrella())
15       .Times(AtLeast(1));
16   const bool its_raining{true};
17   manageUmbrella(its_raining, umbrella);
18 }
```

Compile GTest with ROS 2

- Tests are usually added to a *test* folder within the package;
- Tests can be compiled by adding to the *CMakeLists.txt*:

```
if(BUILD_TESTING)
  find_package(ament_cmake_gtest REQUIRED)

  ament_add_gtest(${PROJECT_NAME}_tutorial_test test/tutorial_test.cpp)
  target_include_directories(${PROJECT_NAME}_tutorial_test PUBLIC
    $<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}/include>
    $<INSTALL_INTERFACE:include>
  )
  target_link_libraries(${PROJECT_NAME}_tutorial_test name_of_local_library)
endif()
```

- If you are using GMocks in your test:

```
find_package(ament_gmock REQUIRED)
ament_add_gmock(${PROJECT_NAME}_mock_test test/mock_test.cpp)
```

- And to the *package.xml:*

```
<test_depend>ament_cmake_gtest</test_depend>
```

Compile GTest with ROS 2

CODE STYLE

- ROS 2 uses the Google C++ Style Guide, with some modifications (check the guidelines!)

- If you want to enforce the code to be formatted following the ROS 2 guidelines:

  CMakeLists.txt:

  ```
  find_package(ament_lint_auto REQUIRED)
  ament_lint_auto_find_test_dependencies()
  ```

  package.xml:

  ```
  <test_depend>ament_lint_auto</test_depend>
  <test_depend>ament_lint_common</test_depend>
  ```

Run Test in ROS 2

- Tests need to be run before committing chenges to the repo!
- To compile and run the tests:
  colcon test --ctest-args tests [package_selection_args]
- To examine the results:
  colcon test-result --all
  colcon test-result --all --verbose (to see test cases which fail)

- If a C++ test is failing, gdb can be used directly on the test executable in the build directory. First, clean the cache and rebuild the code in debug mode:
  colcon build --cmake-clean-cache --mixin debug
- Run the test directly through gdb (C++ debugger):
  gdb -ex run ./build/rcl/test/test_logging
- More info about backtraces and GDB with ROS2 are available in this guide.

Unit Test in ROS 2

- Examples (both in the yasmin and yasmin_ros folders): https://github.com/uleroboticsgroup/yasmin/tree/main

Component Testing in ROS 2

- Unit tests → check individual functions or classes.
- Component tests → check the correct behavior of a ROS 2 node as a whole:
  - Verify that a **node behaves as expected** when integrated into its ROS environment.
  - Focus on **external behavior** (topics, services, actions), not internal implementation details.

How It Works?

1. Launch the **node under test** with its parameters.
2. Replace dependent nodes with **mocks or stubs**.
3. Interact via **ROS interfaces** (publishers, services, actions).
4. Optionally, feed **recorded sensor data** (rosbags) to simulate the robot environment.

Recording and playing back data

A **rosbag** is a file that stores **ROS messages over time**.

- It allows you to **record**, **inspect**, and **replay** the data exchanged between ROS nodes.
- Essential for testing, debugging, and simulation.

Action	Command	Purpose
Record	`ros2 bag record --topics <topic_name>`	Capture specific topics in the system
	`ros2 bag record -a`	Capture all topics in the system
	`ros2 bag record -a -o <bag_name>`	Save bag in specific folder
Play	`ros2 bag play <bag_name>`	Replay recorded messages in real time
	`ros2 bag play -l <bag_name>`	Replay in a loop
Info	`ros2 bag info <bag_name>`	Inspect bag contents (topics, message types, duration)

Component Testing example

- Write a **launch** file that starts the node together with its environment:
- Alongside the node to be tested, we start another launchfile to publish the robot description, and a bagfile containing data previously recorded from the real robot.

```python
#! /usr/bin/env python3

import os
import sys
from ament_index_python import get_package_share_directory
from launch import LaunchDescription
from launch import LaunchService
from launch.actions import ExecuteProcess,
    IncludeLaunchDescription
from launch.launch_description_sources import
    AnyLaunchDescriptionSource
from launch_ros.actions import Node
from launch_testing.legacy import LaunchTestService


def generate_launch_description():
    # Assuming the bagfile to be installed in the shared
    folder
    bagfile = get_package_share_directory('a_package') + "/
    bags/" + "my_bag"
    return LaunchDescription([
        IncludeLaunchDescription(
            AnyLaunchDescriptionSource(
                os.path.join(
                    robot_description_pkg,
                    'launch/robot_description.launch.py'))
        ),
        Node(
            package='a_package',
            executable='name_of_the_exe_to_be_tested',
            name='test_node_name',
            output='screen'
        ),
        ExecuteProcess(
            cmd=['ros2', 'bag', 'play', bagfile,
                 '-l', '--clock'
        )
    ])


def main(argv=sys.argv[1:]):
    ld = generate_launch_description()

    testExecutable = os.getenv('TEST_EXECUTABLE')

    first_test_action = ExecuteProcess(
        cmd=[testExecutable],
        name='your_test_name',
        output='screen')

    lts = LaunchTestService()
    lts.add_test_action(ld, first_test_action)
    ls = LaunchService(argv=argv)
    ls.include_launch_description(ld)
    return lts.run(ls)

if __name__ == '__main__':
    sys.exit(main())
```

Component Testing example

The content of the interchanged data can be compared to the expected one using the usual assertions provided by **Gtests**;

An action server residing on the tested node is called by an action client created in the test.

```cpp
#include <gtest/gtest.h>
#include <chrono>
#include <memory>
#include "my_msgs/action/my_action.hpp"
#include "rclcpp/rclcpp.hpp"
#include "rclcpp_action/rclcpp_action.hpp"

using namespace testing;
class MyTestFixture : public ::testing::Test {
 public:
  static void SetUpTestCase() { rclcpp::init(0, nullptr); }

  void SetUp() override {
    node_ = rclcpp::Node::make_shared("test_node");
    client_ptr_ = rclcpp_action::create_client<MyAction>(
    test_node_, "my_action");
  }

  void sendGoalToNode(const my_msgs::action::MyAction::Goal
    & goal, rclcpp_action::ClientGoalHandle<MyTestFixture::
    MyAction>::WrappedResult& result) {
    auto fut_response = client_ptr_->async_send_goal(
    goal_msg);
    ASSERT_TRUE(rclcpp::spin_until_future_complete(
    test_node_, fut_response) ==
                rclcpp::FutureReturnCode::SUCCESS);
    auto goal_handle = fut_response.get();
    auto fut_result = client_ptr_->async_get_result(
    goal_handle);
    ASSERT_TRUE(rclcpp::spin_until_future_complete(
    test_node_, fut_result) ==
                rclcpp::FutureReturnCode::SUCCESS);
    result = fut_result.get();
  }

  std::shared_ptr<rclcpp::Node> test_node_;
  rclcpp_action::Client<my_msgs::action::MyAction>::
    SharedPtr client_ptr_;
};

TEST_F(MyTestFixture, TestMyAction) {
  ASSERT_TRUE(client_ptr_->wait_for_action_server(5s));
  auto goal = my_msgs::action::MyAction::Goal();
  rclcpp_action::ClientGoalHandle<MyTestFixture::MyAction
    >::WrappedResult goal_result;
  sendGoalToNode(goal_msg, goal_result);
  EXPECT_TRUE(result->success);
}

int main(int argc, char** argv) {
  ::testing::InitGoogleTest(&argc, argv);
  return RUN_ALL_TESTS();
}
```

Component Testing example

The same fixture class can of course be used to define multiple tests. These tests will then behave as unit tests, with the only difference that a node, executed by the previously defined launchfile, will be running in parallel to them. To achieve this, we need to add a couple of entries to our CMakeLists.txt

In case you need the gmock features, you might have to additionally add the following before ament_add_test:

```
find_package(ament_cmake_gmock REQUIRED)
_ament_cmake_gmock_find_gmock()

target_include_directories(${TEST_NAME} PUBLIC ${
    GMOCK_INCLUDE_DIRS})
target_link_libraries(${TEST_NAME}
  gtest_main
  gmock
)
```

```
find_package(ament_cmake_gtest REQUIRED)
include(GoogleTest)
SET (TEST_NAME "name_of_your_test")

ament_add_gtest_executable(${TEST_NAME}
  test_file.cpp
)

target_link_libraries(${TEST_NAME}
  gtest_main
)

ament_target_dependencies(${TEST_NAME}
  rclcpp
)

ament_add_test(${TEST_NAME}
  GENERATE_RESULT_FOR_RETURN_CODE_ZERO
  COMMAND "${CMAKE_CURRENT_SOURCE_DIR}/launchfile_name.py"
  WORKING_DIRECTORY "${CMAKE_CURRENT_BINARY_DIR}"
  ENV
    TEST_DIR=${TEST_DIR}
    TEST_LAUNCH_DIR=${TEST_LAUNCH_DIR}
    TEST_EXECUTABLE=$<TARGET_FILE:${TEST_NAME}>
)
```

Integration test

- Ensure that **multiple ROS 2 nodes** (a subsystem) can **work together correctly** to achieve a shared goal.
- Goes beyond component tests: checks **inter-node cooperation**, not single-node logic.

1. **Create a launch file**
   Start all nodes under test and their environment (parameters, rosbag, etc.).
2. **Write integration tests**
   Interact with one or more nodes via their ROS interfaces (topics, services, actions).
3. **Verify expected results**
   Check that joint behavior matches the desired system outcome.

Simulation-Based Test

- As systems grow larger, **integration tests** become complex and fragile. Simulators offer a way to test **whole robot systems** — without physical hardware or complex setup.
- The simulator models the **robot**, its **sensors**, and the **environment**.
- Nodes interact as if connected to real hardware.
- Predict robot behavior **accurately and safely.**
- Test in **diverse, repeatable environments** (even randomized).
- Integrate smoothly into **CI/CD pipelines.**

Testing on Hardware

Even with perfect simulations, **new issues often emerge** in the physical world.

Challenge	Description
Hardware can fail	Broken circuits, loose joints, scratched sensors: always verify the hardware first.
Communication latency	Real buses introduce unpredictable delays, critical for control loops.
Hardware degrades	Wear, drift, dirt, and calibration errors accumulate over time.
Sensor noise & artefacts	Shadows, reflections, sunlight, or interference can affect readings.
Unpredictable environments	Surfaces, objects, and lighting change, impacting robot performance.

Best Practices

- Test **hardware functionality** before software.
- Calibrate and log **latency and drift** regularly.
- Keep a **hardware checklist** for systematic verification.
- Use simulations and rosbags first, then move to hardware tests gradually.