UNIVERSITÀ
DI TRENTO
Department of
Information Engineering and Computer Science

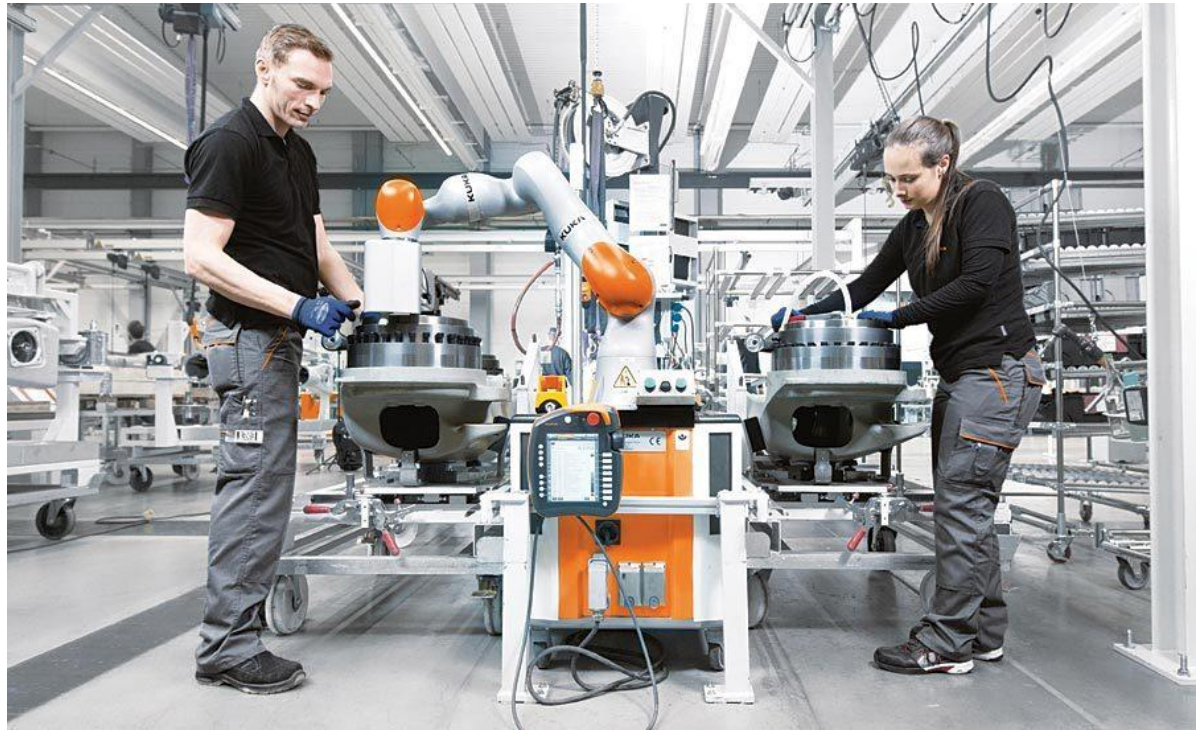
Robot Programming

Edoardo Lamon

Software Development for Collaborative Robotics

Academic Year 2025/26

Industrial vs Collaborative Robotics



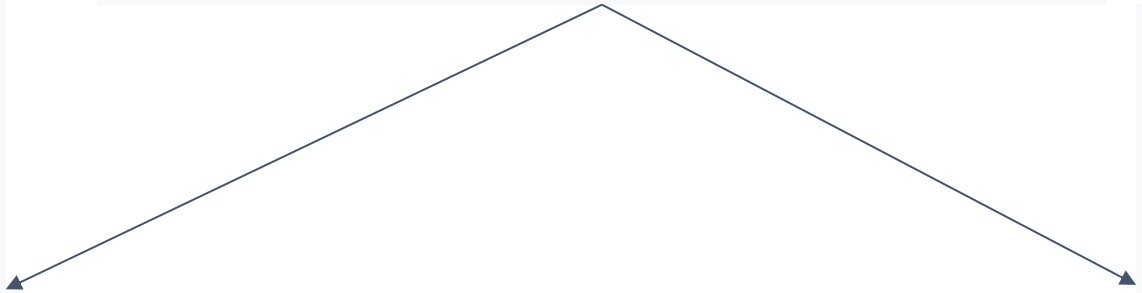
The World of Industrial Robotics

- "**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

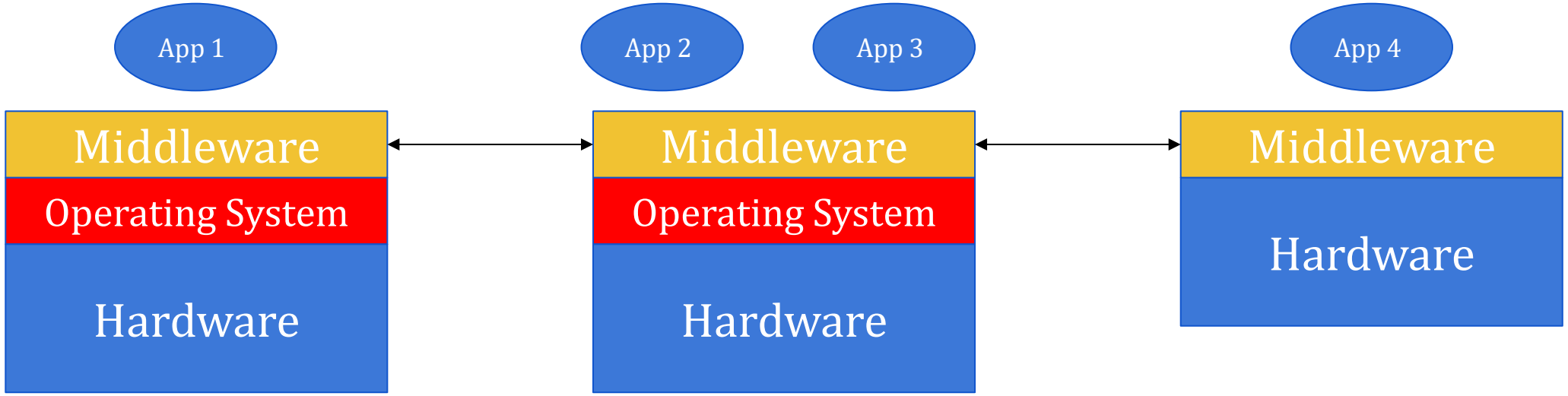
- **Robot-to-robot** communication
- External services in the **cloud** (e.g., for strategic decisions)
- **Heterogeneous** types of hardware
  - Microcontrollers
  - GPU
  - Industrial PC
- **Timing constraints** on computations
  - Particularly true for unstable systems like drones or legged robots
- 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:

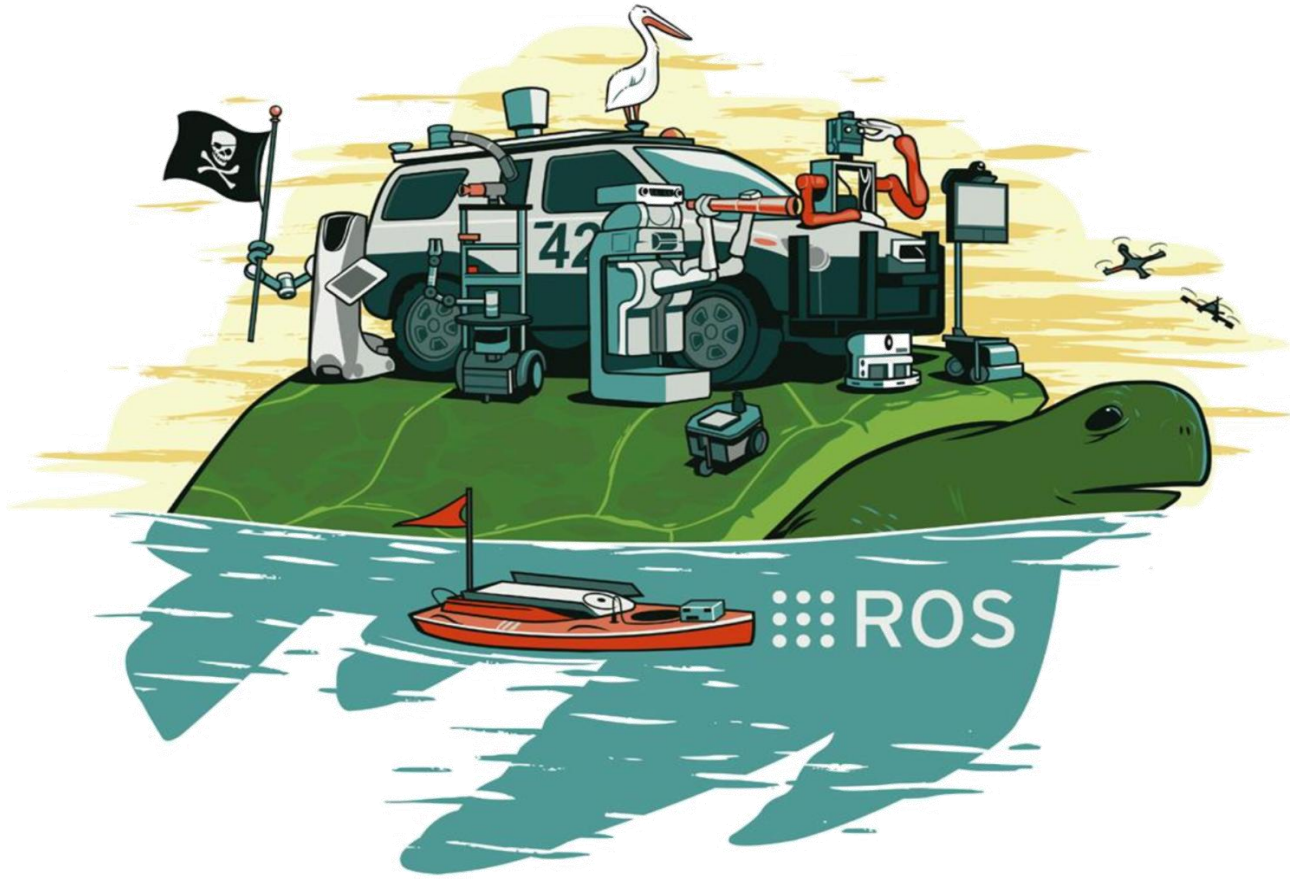
Type	Services	Examples
Message Oriented Middleware	Receiving and sending of messages over distributed applications	Amazon Simple Notification System (SNS), IBM MQ, Amazon AWS IoT Core
Remote Procedure Call (RPC)	Calling procedures on remote systems and performing synchronous or asynchronous interactions	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

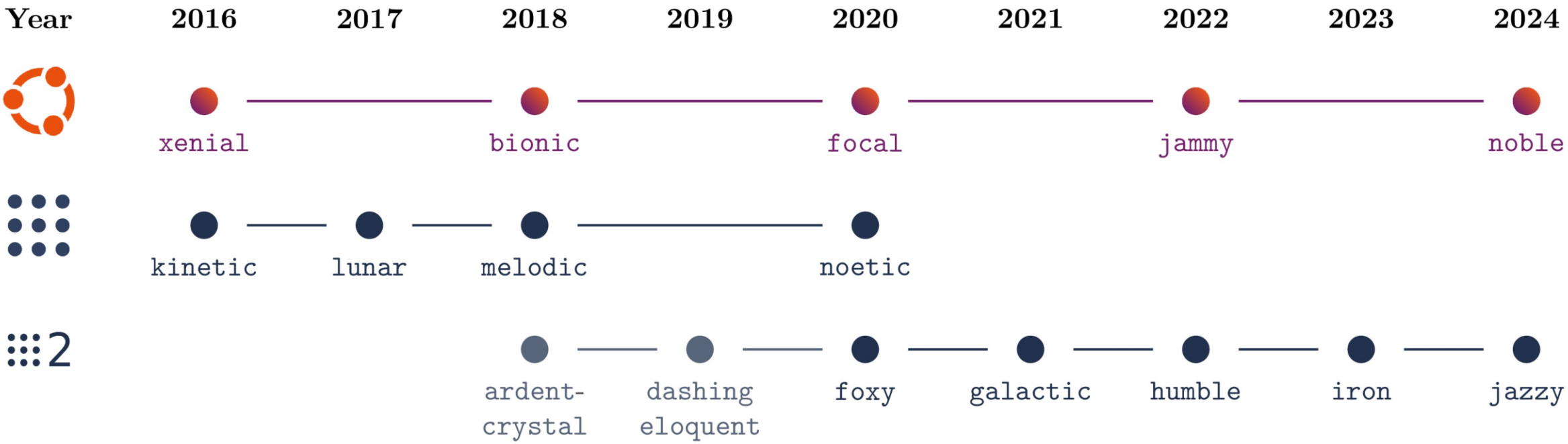


(Image taken from Willow Garage's "What is ROS?" presentation)

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.

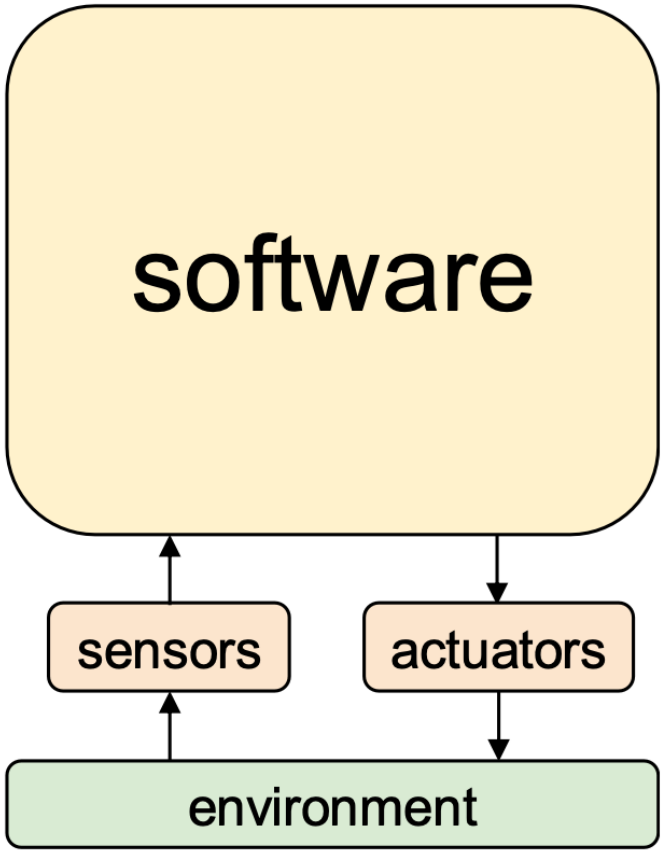
Transition to ROS 2

- 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?

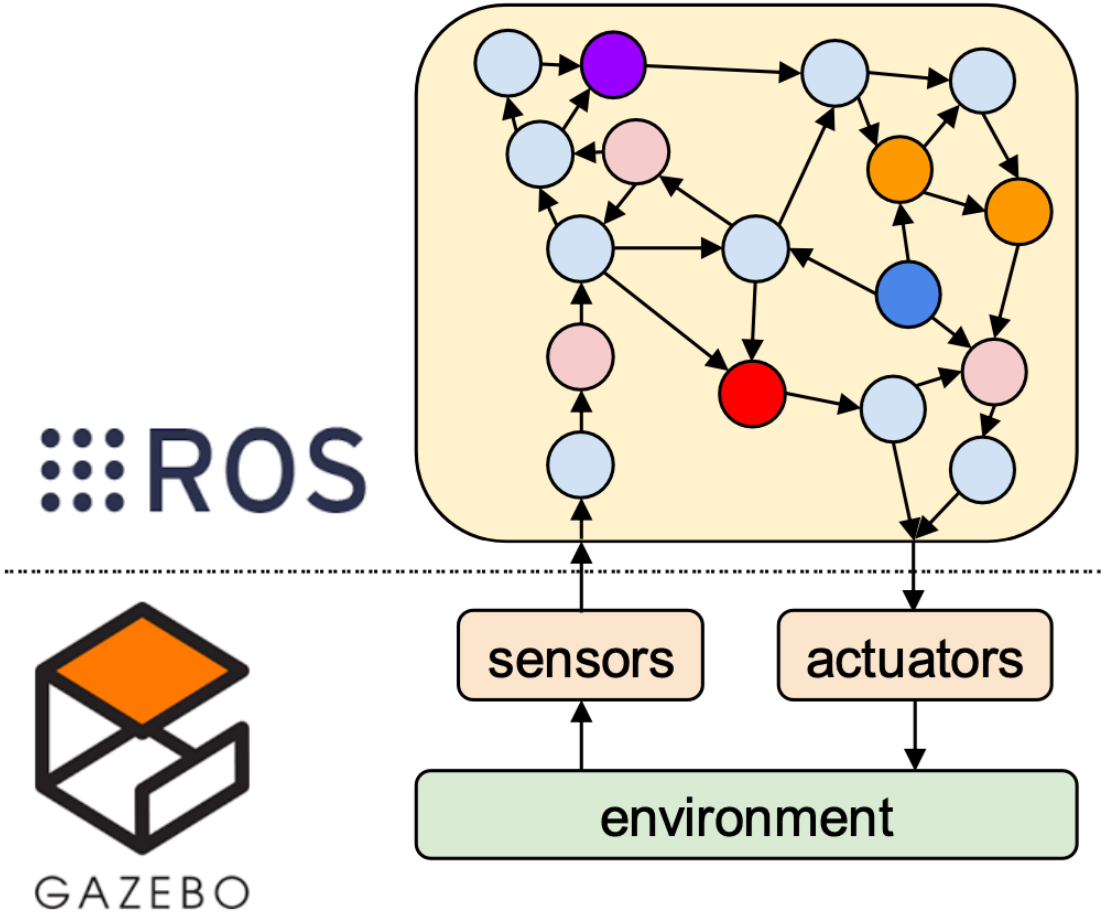
Software connecting sensors to actuators to interact with the environment

(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.

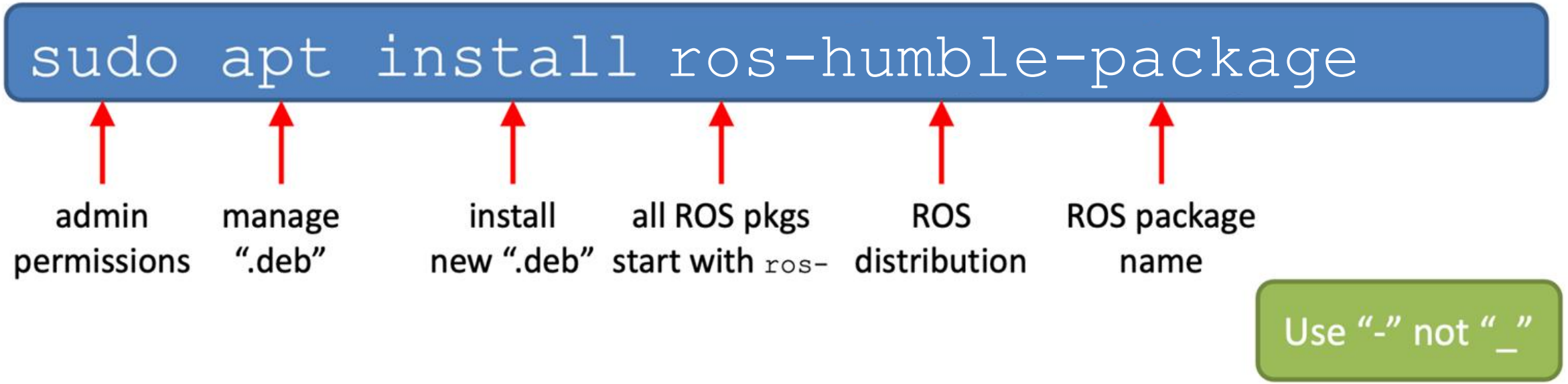
ROS 2 Humble Installation

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)

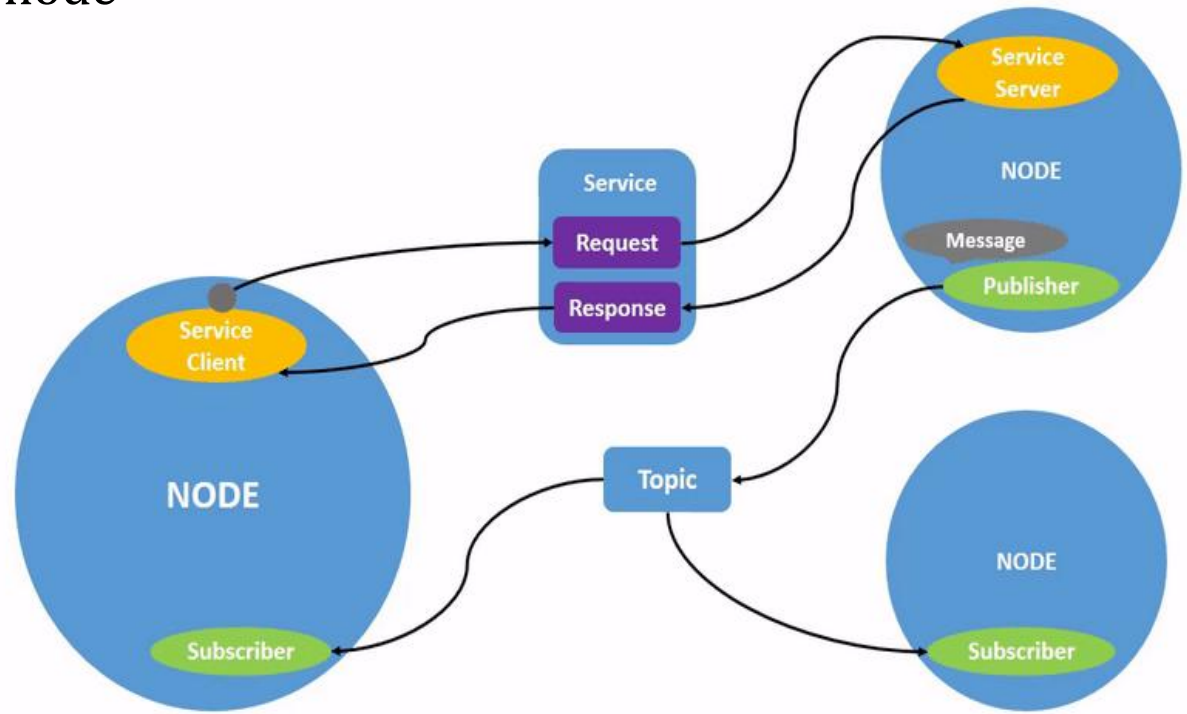
ROS 2 Nodes

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

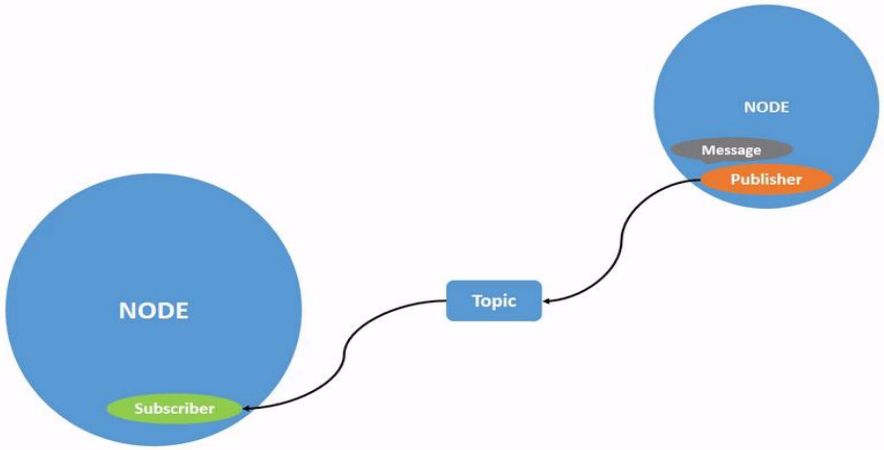
- There are four ways to communicate between nodes in ROS2
  - **Streaming topics**
  - **Remote Procedure Call (RPC) Services**
  - **Actions**

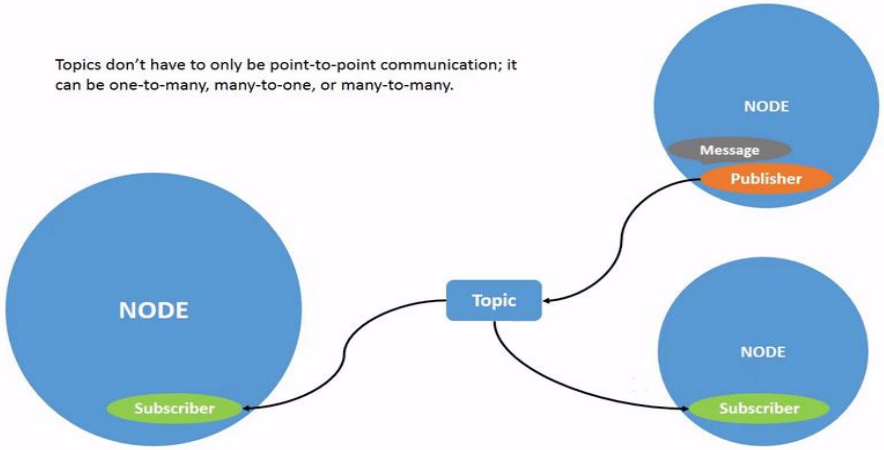
- In addition, to configure nodes at startup (and during runtime) without changing the code, the **parameter server** is used

- 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

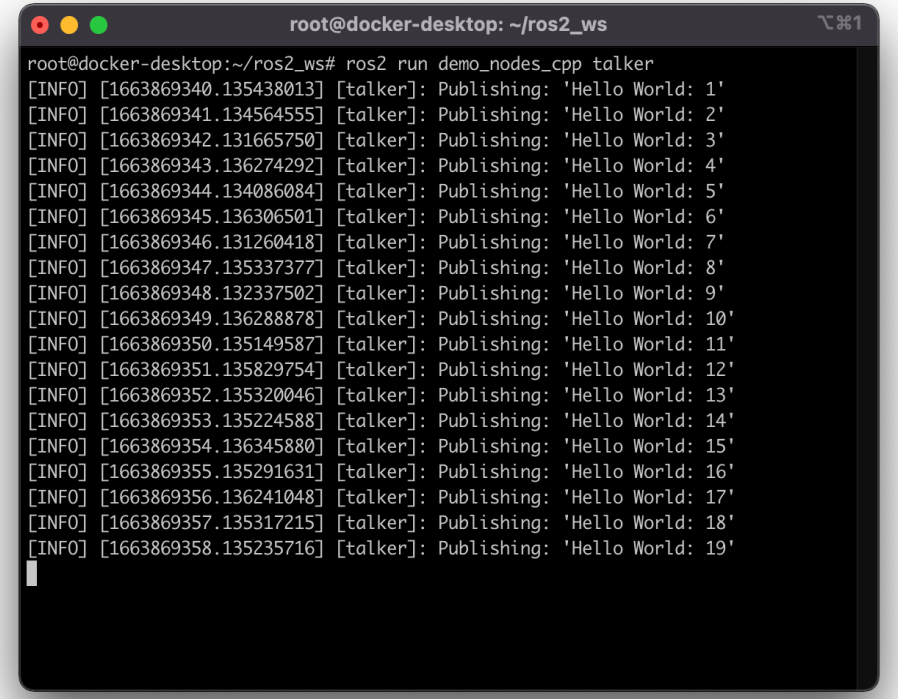
  std_msgs/msg/String.msg

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.

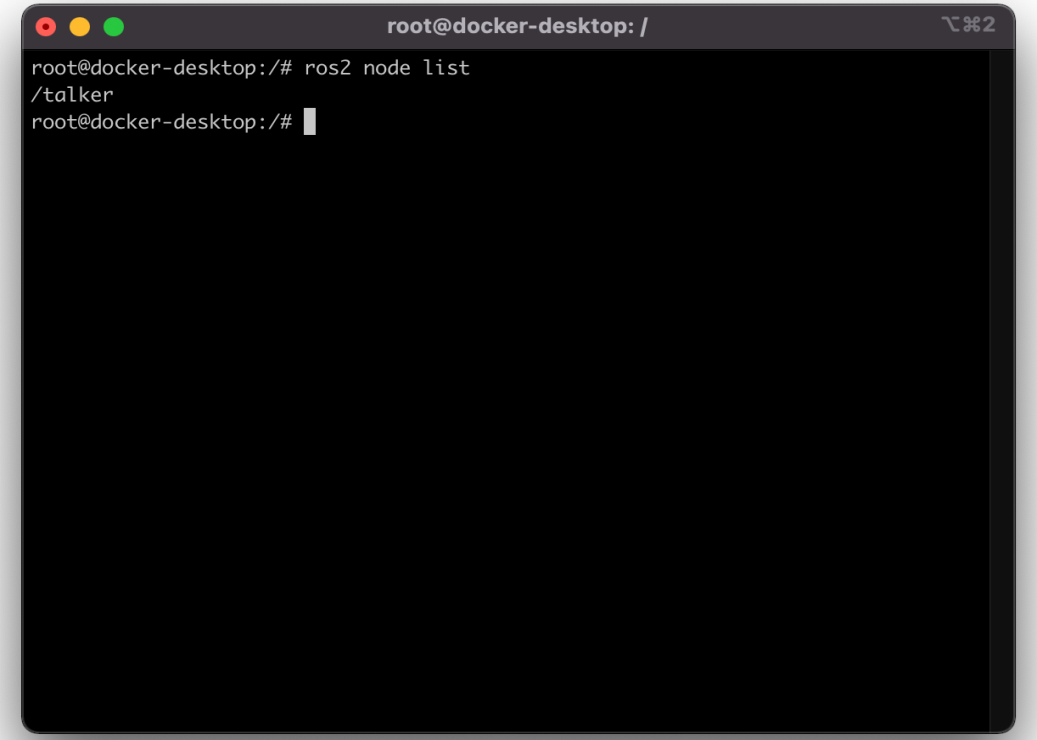
Example – Nodes & Topics

$ ros2 run demo_nodes_cpp talker



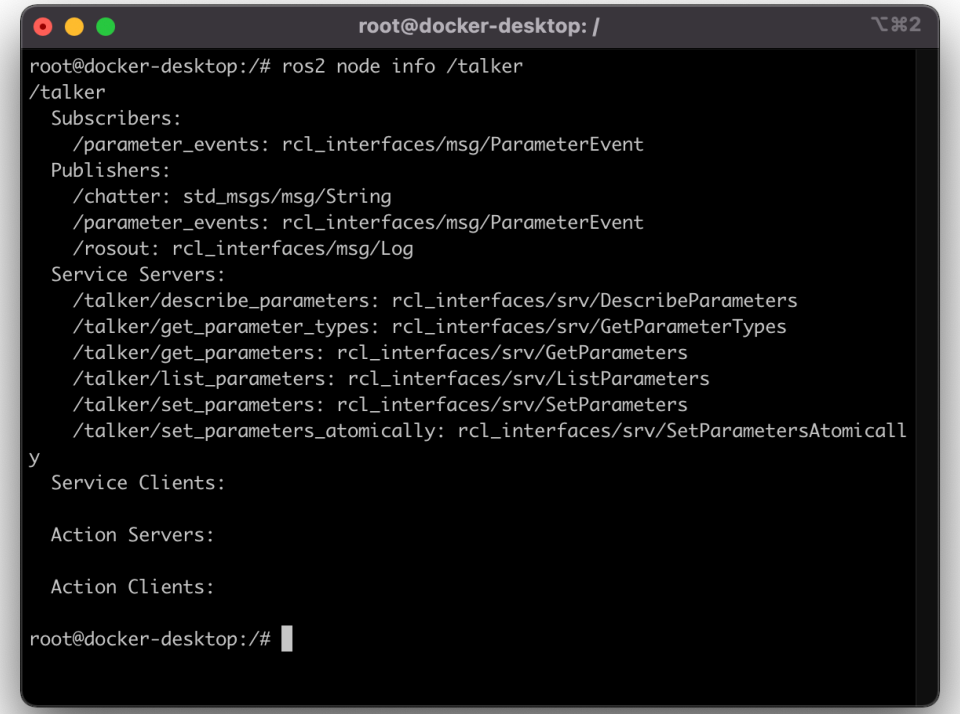
Example – Nodes & Topics

$ ros2 node list



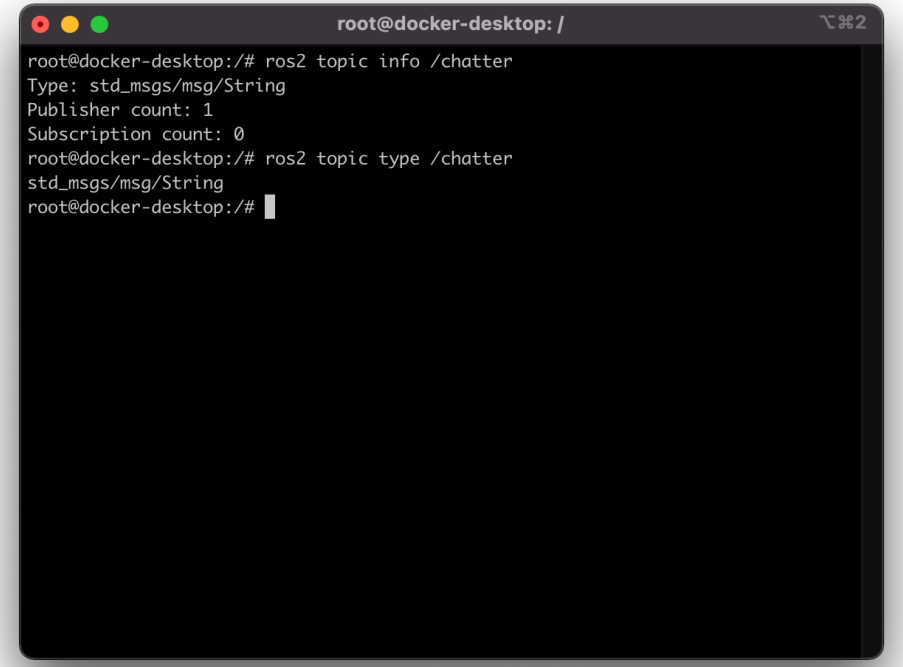
Example – Nodes & Topics

$ ros2 node info /talker



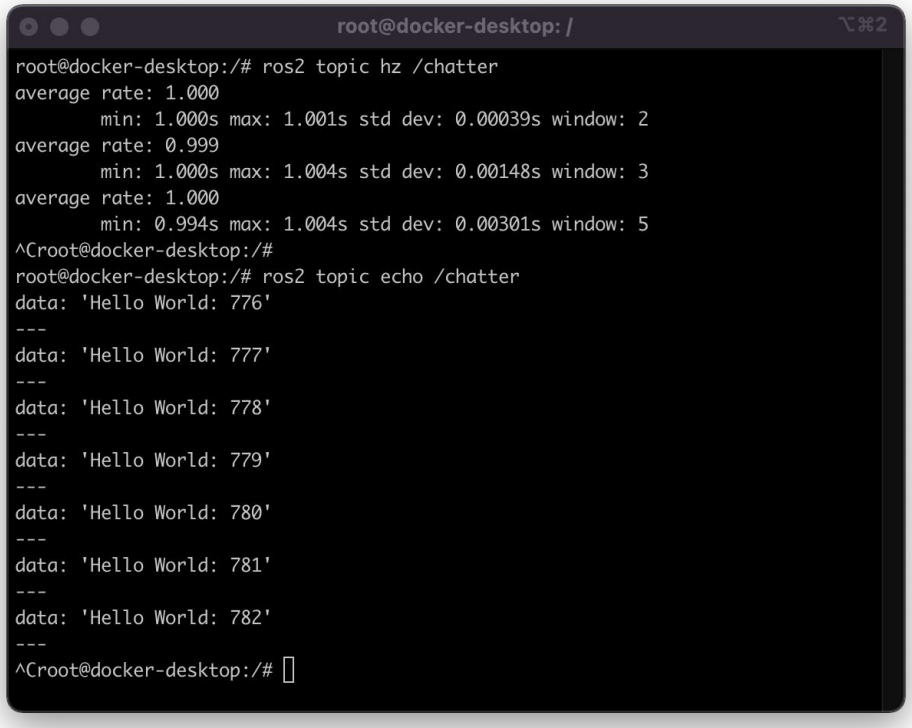
Example – Nodes & Topics

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



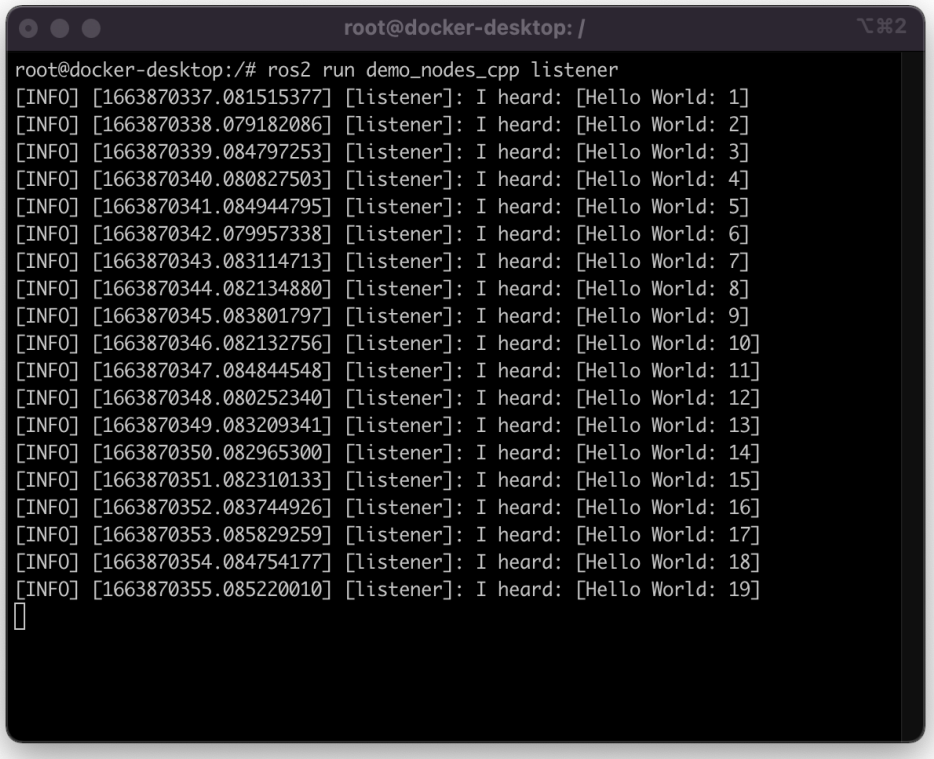
Example – Nodes & Topics

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



Example – Nodes & Topics

$ ros2 run demo_nodes_cpp listener

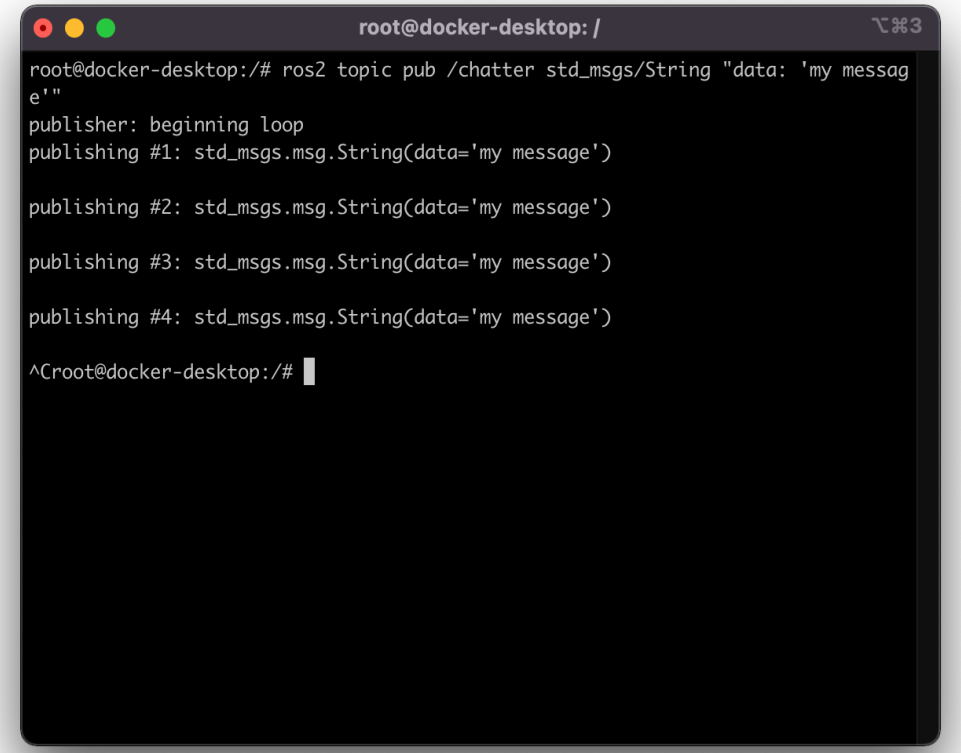


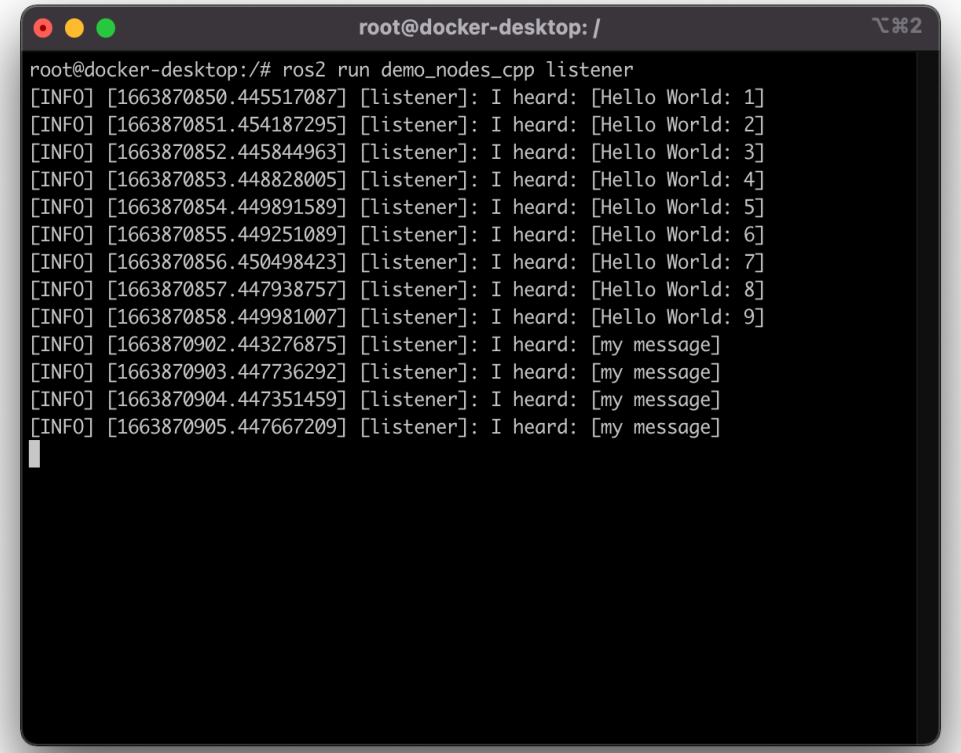
Example – Nodes & Topics

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

Example – Nodes & Topics

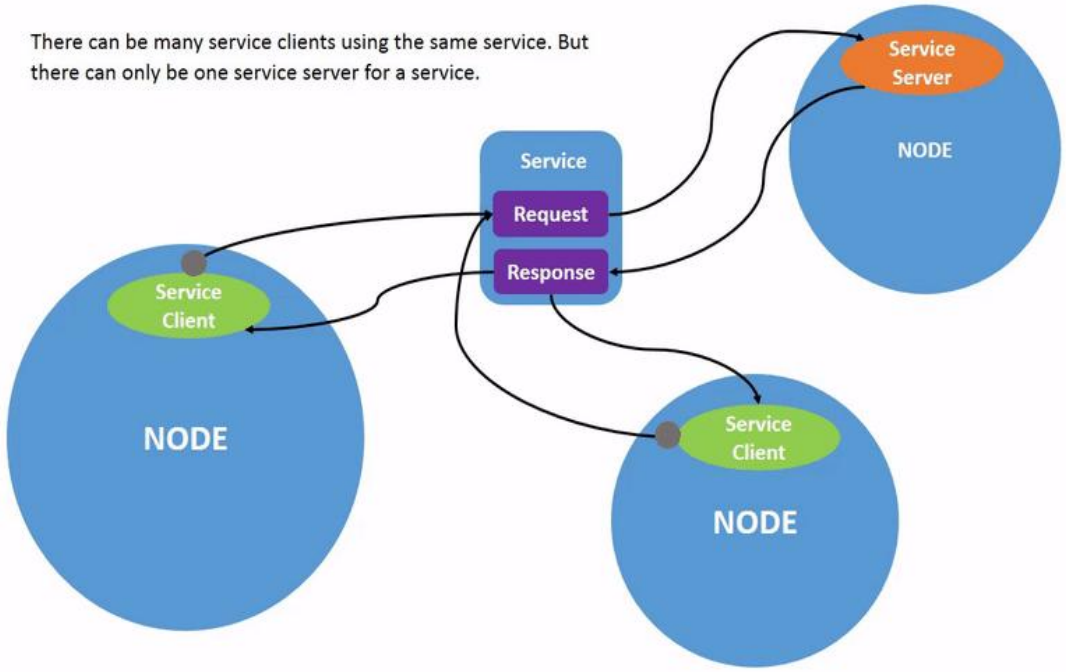
$ ros2 topic pub /chatter std_msgs/String "data: 'my message'"
$ ros2 topic pub [--once] [-t times] [-r rate] ...





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

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

- The service interface is defined in a **.srv** file (same data structures supported as .msg)

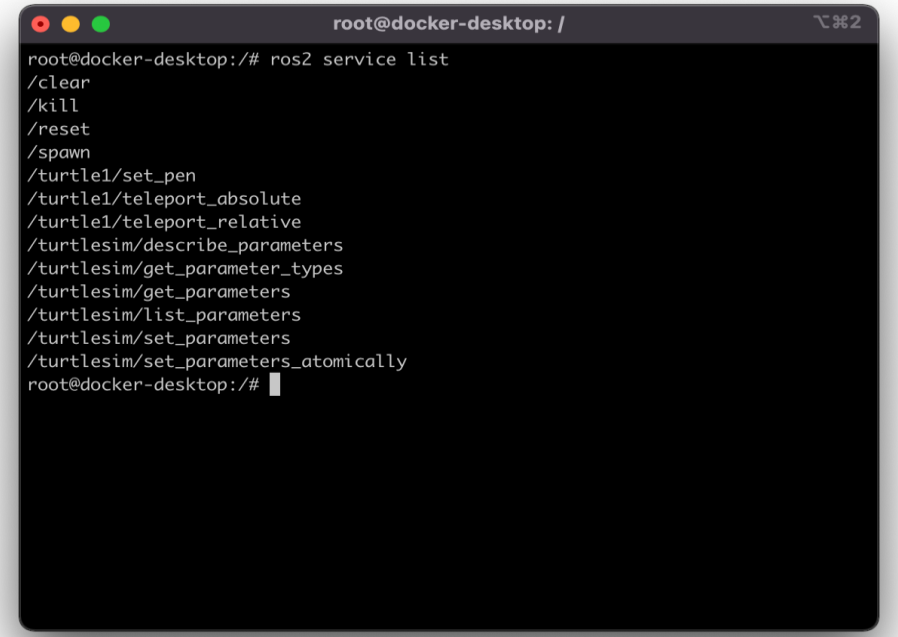
- It is possible to create **custom** .srv

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

- 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.

Example – Services

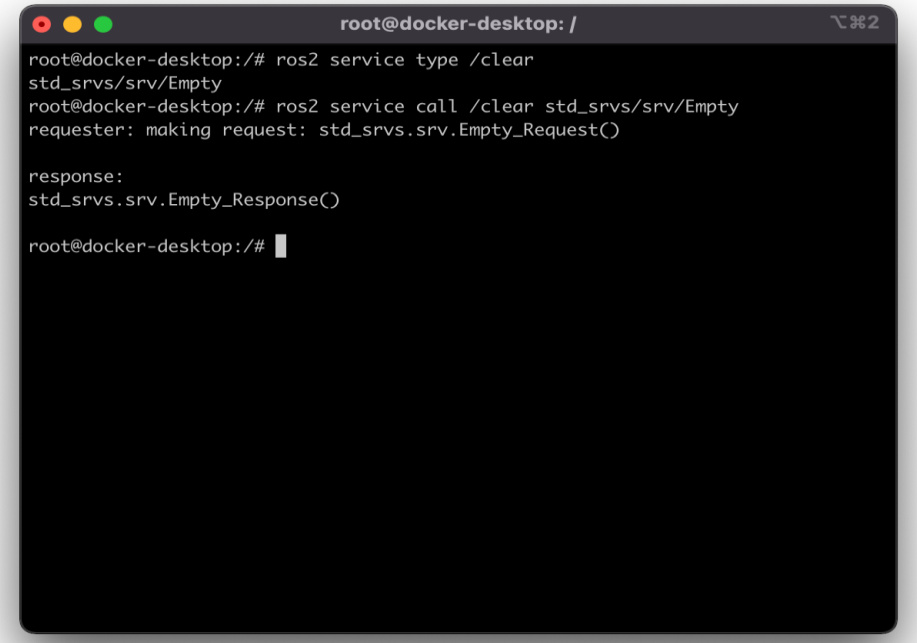
$ ros2 run turtlesim turtlesim_node
$ ros2 service list

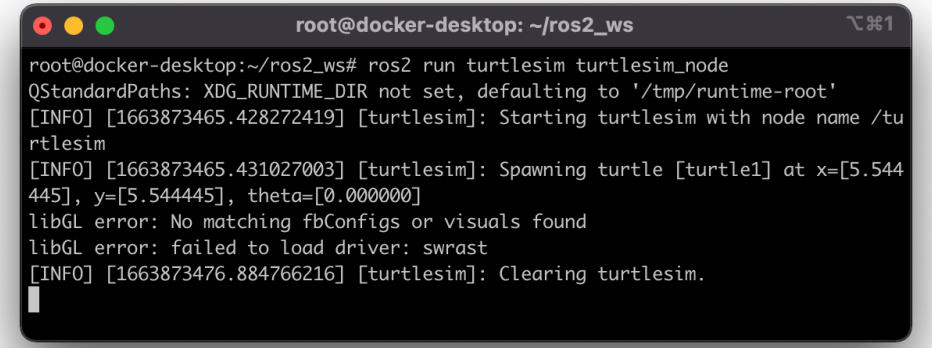


$ ros2 run turtlesim turtle_teleop_key

Example – Services

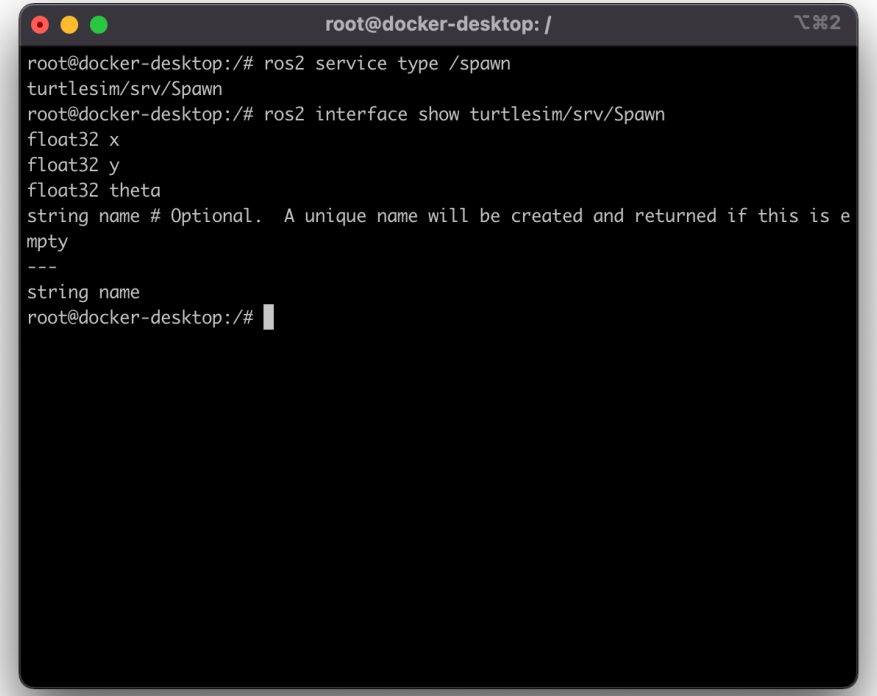
$ ros2 service type /clear
$ ros2 service call /clear std_srvs/srv/Empty





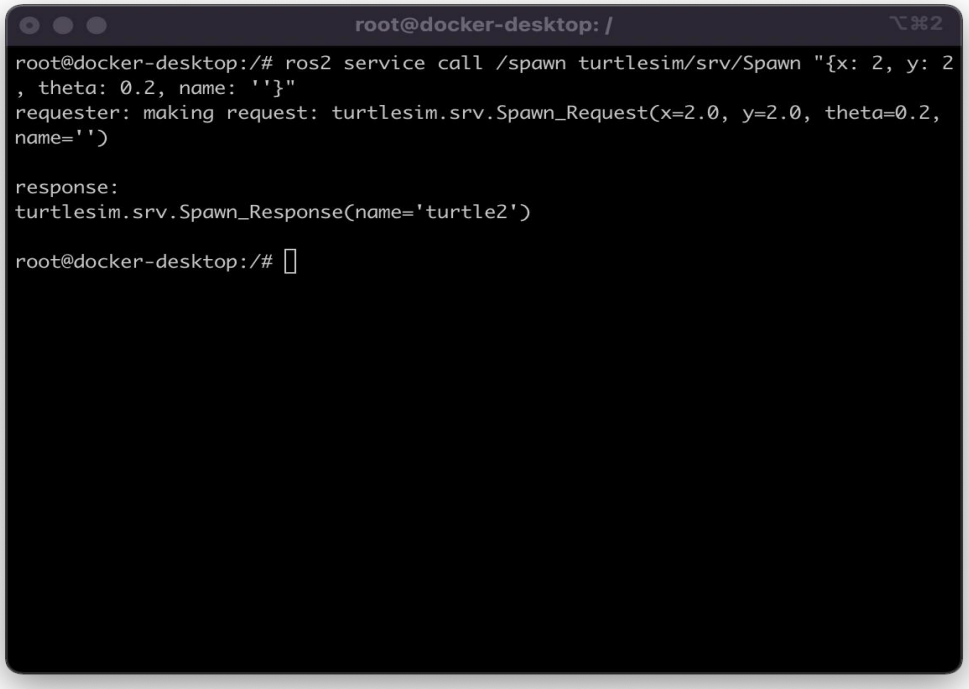
Example – Services

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



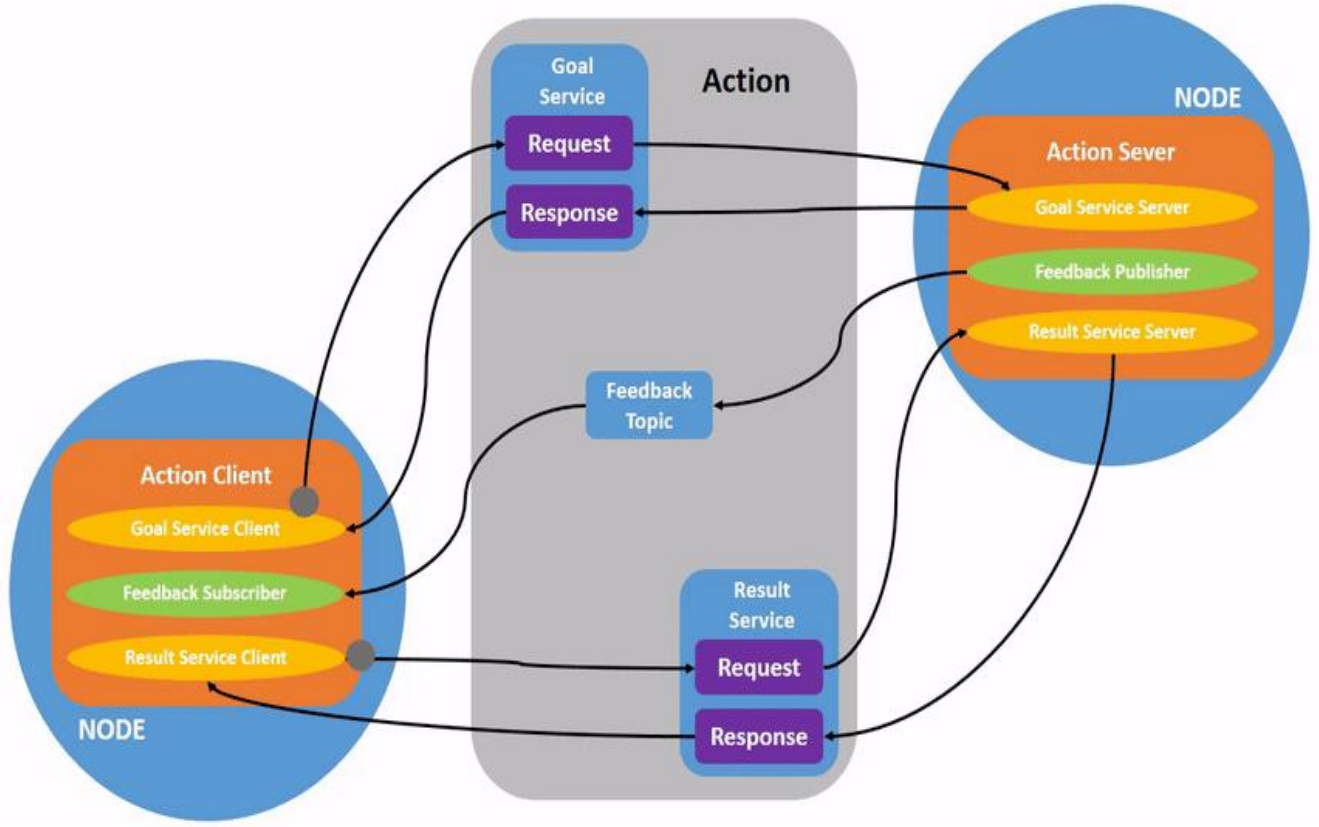
Example – Services

$ 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
```

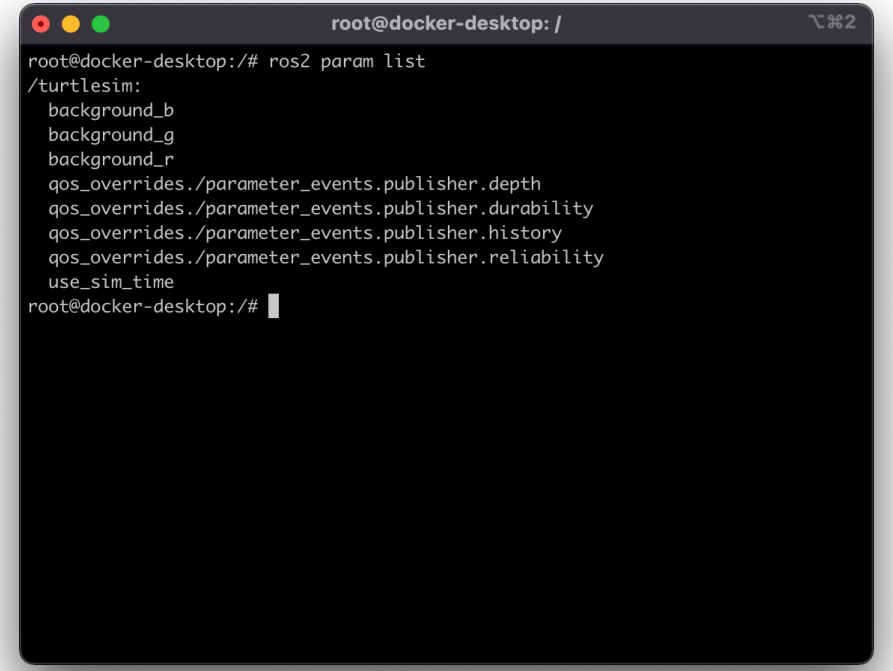
ROS 2 Parameters

- 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**

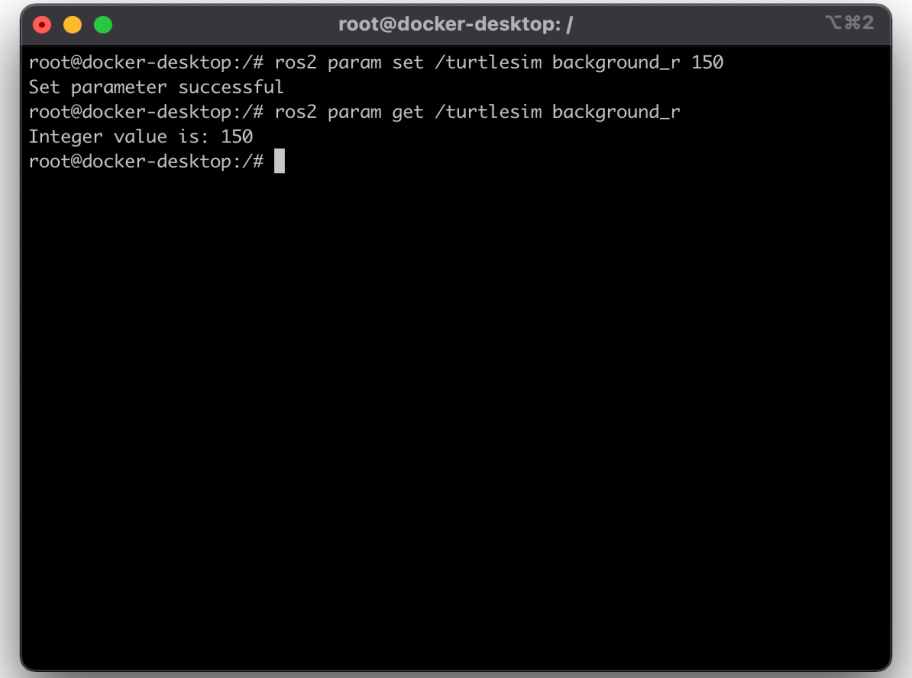
$ ros2 param list

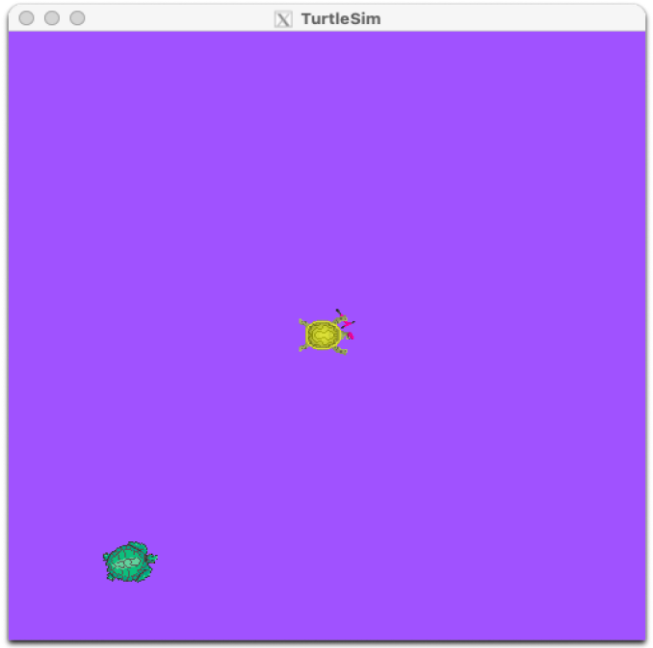


Example – Parameters

$ ros2 param set /turtlesim background_r 150

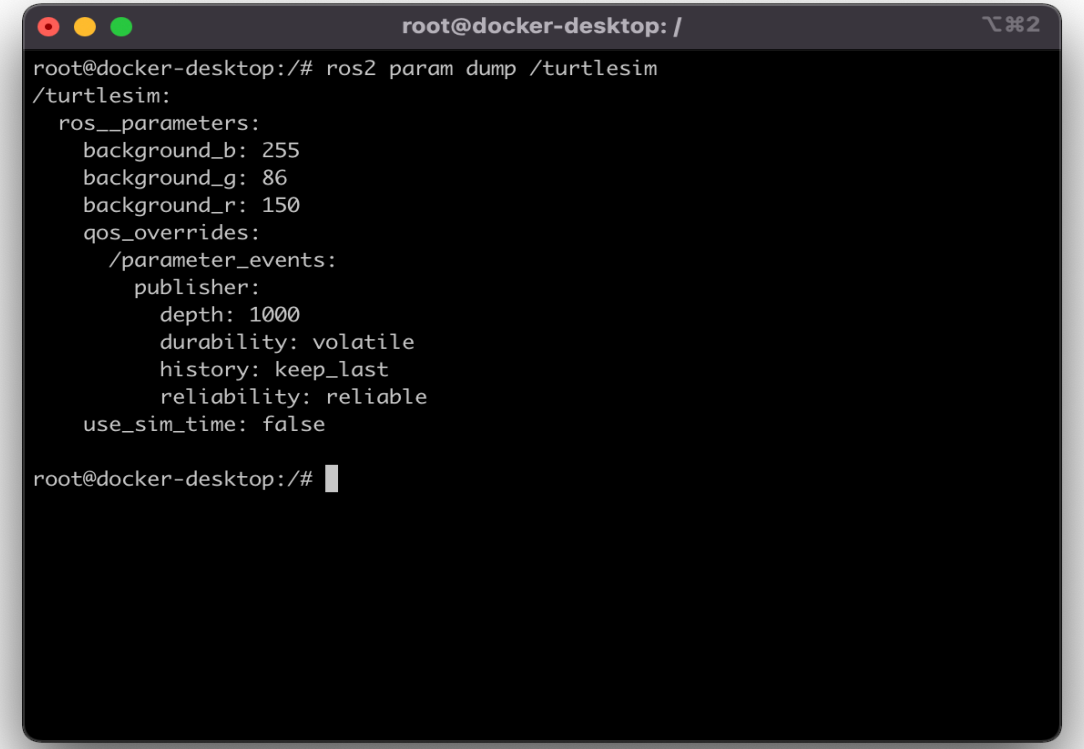
$ ros2 param get /turtlesim background_r





Example – Parameters

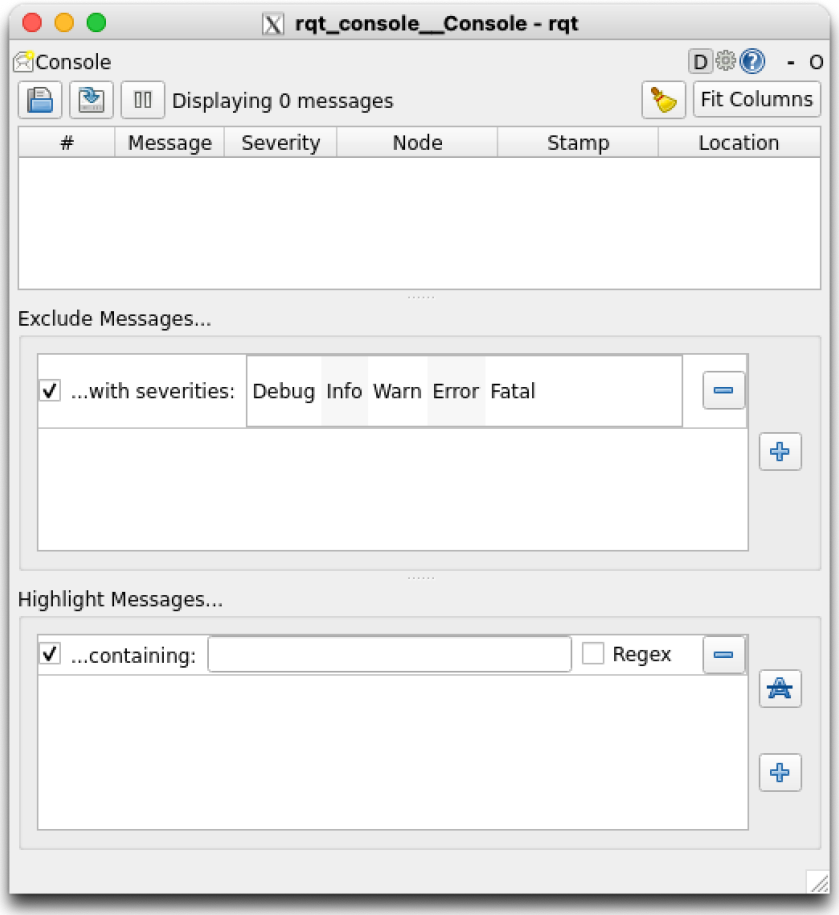
$ ros2 param dump /turtlesim
$ ros2 param dump /turtlesim > turtlesim.yaml
$ ros2 param load /turtlesim turtlesim.yaml



ROS 2 Debugging

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

ros2 run rqt_console rqt_console

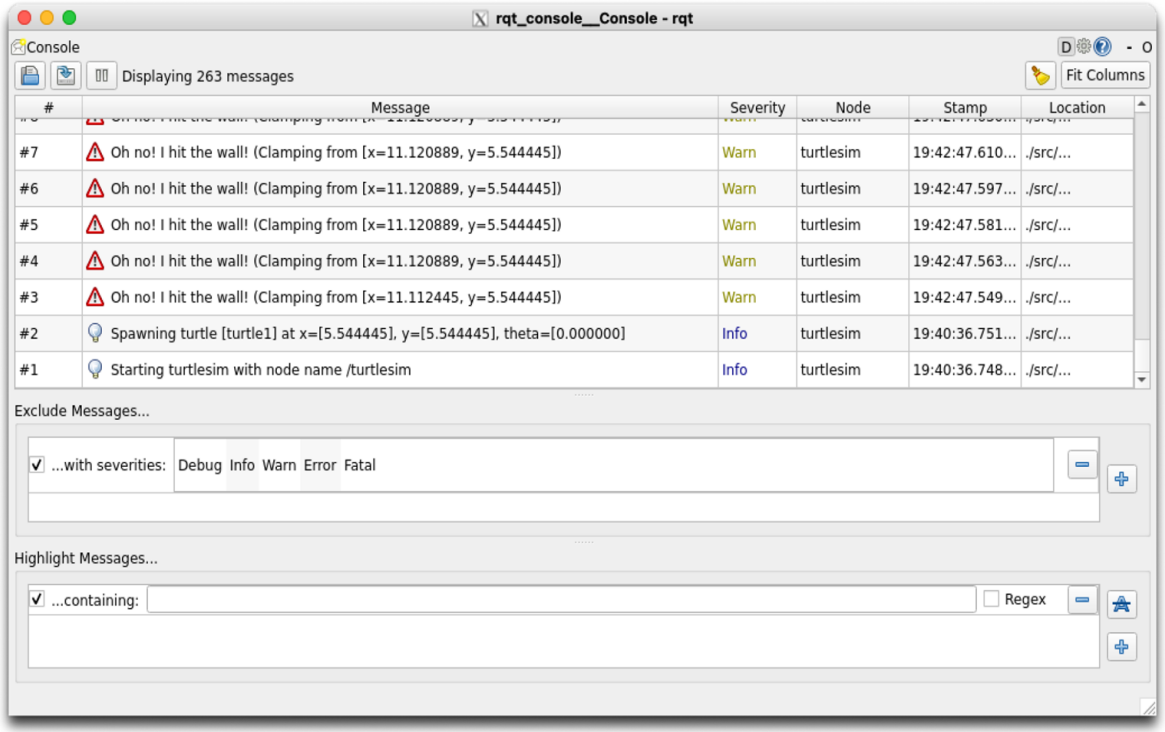


ROS 2 Debugging

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.