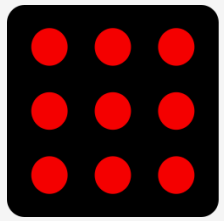
INTRO TO ROS 2 & ROBOT PROGRAMMING

Fast track to robot development



Intelligent Robotics Lab

Universidad Rey Juan Carlos

José Miguel Guerrero Hernández

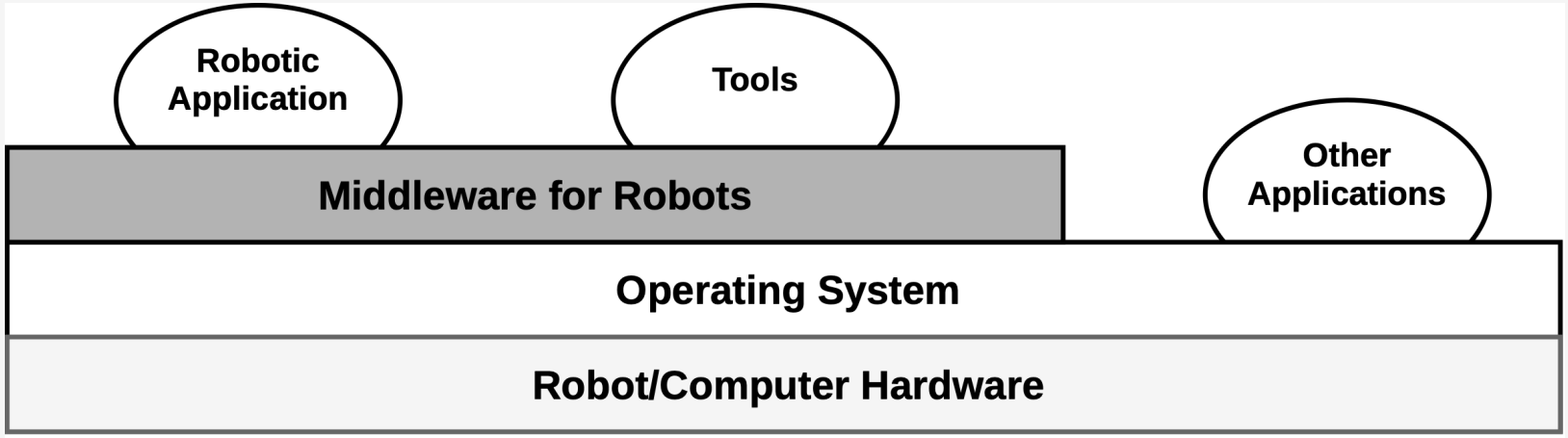
josemiguel.guerrero@urjc.es

Objectives

- Understand the purpose of ROS 2 and its role in modern robotic systems
- Identify the core components of a ROS 2 system: nodes, topics
- Grasp the ROS 2 communication model, including publish/subscribe
- Visualize the architecture of a ROS 2-based robotic system as a graph
- Appreciate the advantages of ROS 2
- Build foundational knowledge to interpret and design basic ROS 2 systems

Programming robots:

- Robots need to be programmed to perform useful tasks
- Middleware simplifies development by providing drivers, libraries, and tools
- Most middleware was robot-specific and did not expand beyond initial labs
- ROS stands out due to its large, active global community, making it widely adopted and reusable



ROS (Robot Operating System):

"The Robot Operating System (ROS) is a set of software libraries and tools that help you build robot applications. From drivers to state-of-the-art algorithms, and with powerful developer tools, ROS has what you need for your next robotics project. And it's all open source."

http://www.ros.org/

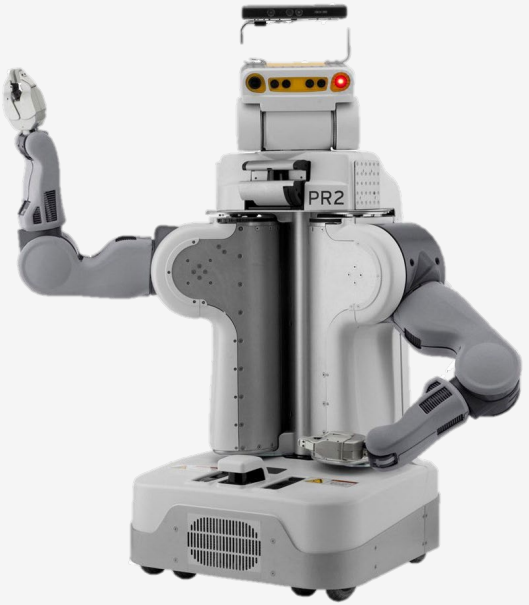
ROS history:

- 2006: Started at Stanford's STAIR project
- 2007: First code released at Willow Garage
- 2010: ROS 1.0 officially released with PR2 robot
- 2013: OSRF took over ROS development
- Now: Widely adopted open-source standard in robotics

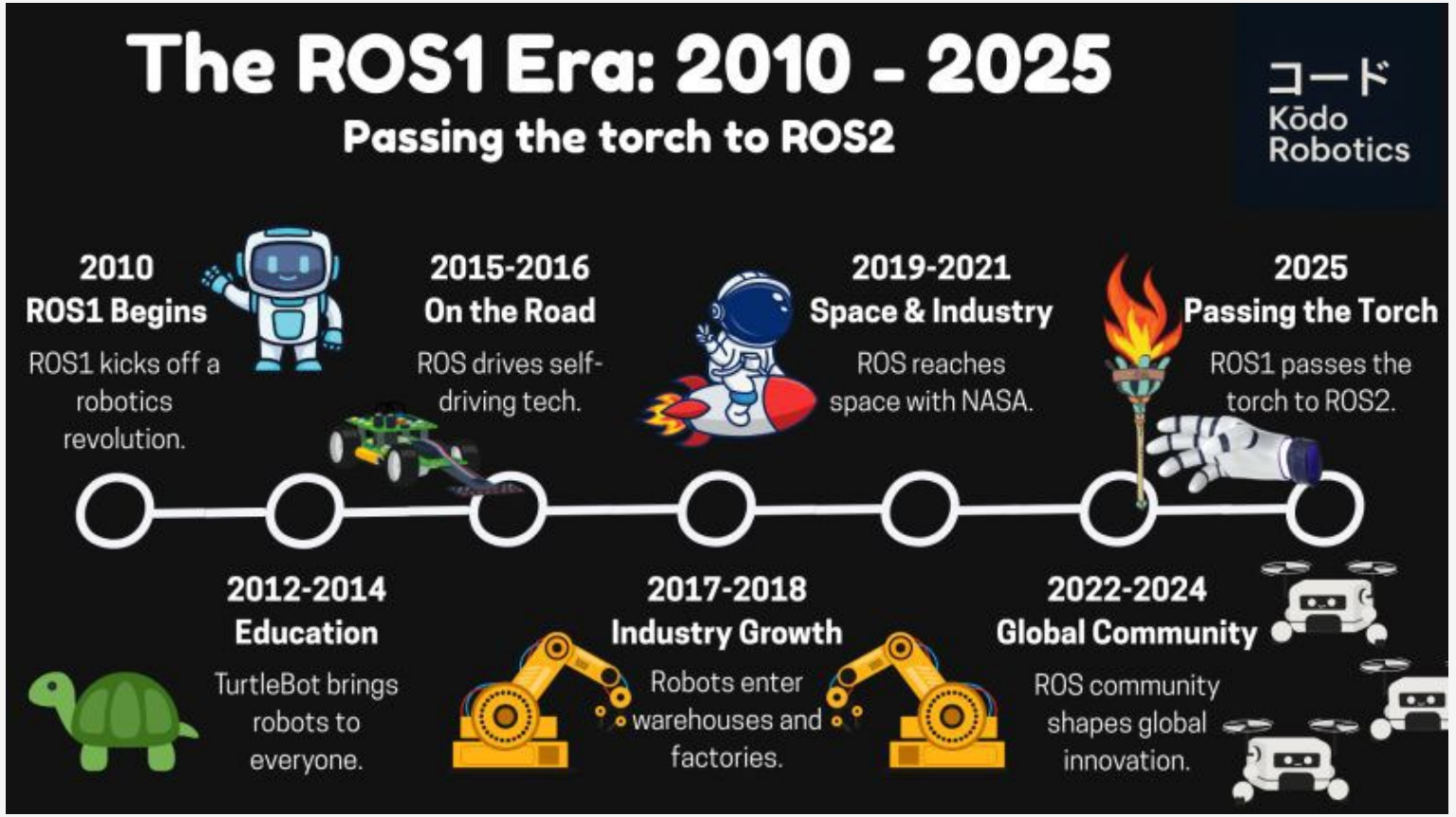
Willow Garage

Box Turtle

Open Source Robotics Foundation

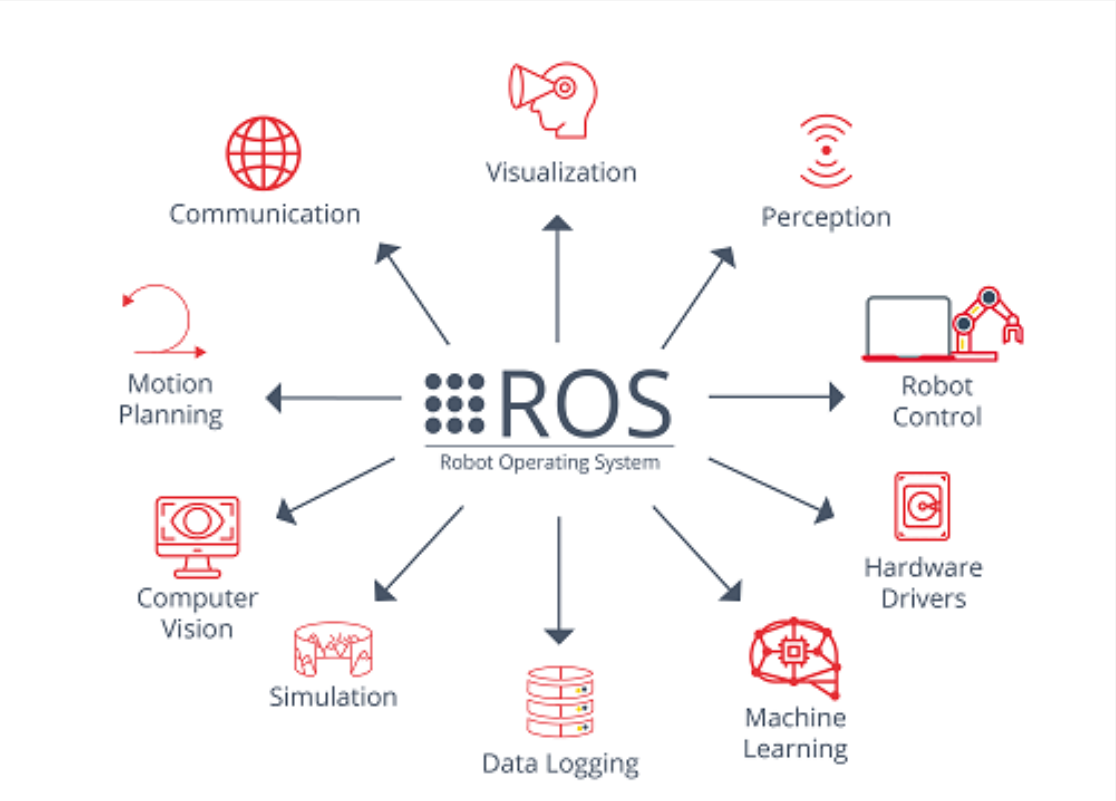


ROS history:


The ROS1 Era: 2010 - 2025
Passing the torch to ROS2
コード
Kōdo
Robotics
2010
ROS1 Begins
ROS1 kicks off a robotics revolution.
2012-2014
Education
TurtleBot brings robots to everyone.
2015-2016
On the Road
ROS drives self-driving tech.
2017-2018
Industry Growth
Robots enter warehouses and factories.
2019-2021
Space & Industry
ROS reaches space with NASA.
2022-2024
Global Community
ROS community shapes global innovation.
2025
Passing the Torch
ROS1 passes the torch to ROS2.

What is ROS 2?

- Successor to ROS 1 with industry-grade communication
- Open-source OS for modular, scalable robotics
- Easily integrates diverse sensors and data formats
- Cross-platform: Linux, Windows, macOS, microcontrollers
- Supports real-time, distributed, and secure systems



ROS 1 vs ROS 2:

- Communication: ROS 2 uses DDS middleware, replacing the ROS 1 master node
- Real-time support: Built-in in ROS 2, limited or hacked in ROS 1
- Multi-robot support: Native in ROS 2, complex in ROS 1
- Security: ROS 2 includes encryption and authentication (DDS-Security)
- Launch system: ROS 2 uses Python-based launch with improved flexibility
- Active development: ROS 2 is the focus of all future ROS tooling and libraries
- Backward compatibility: ROS 2 is not directly compatible with ROS 1, but bridges exist

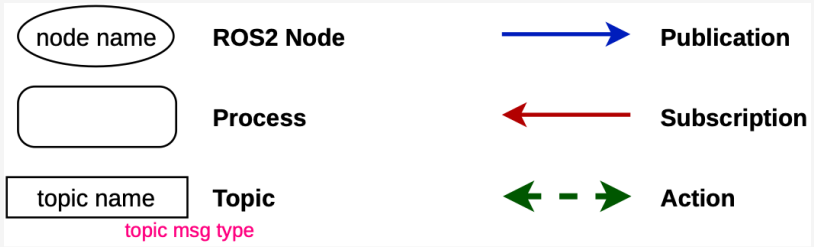
ROS 2

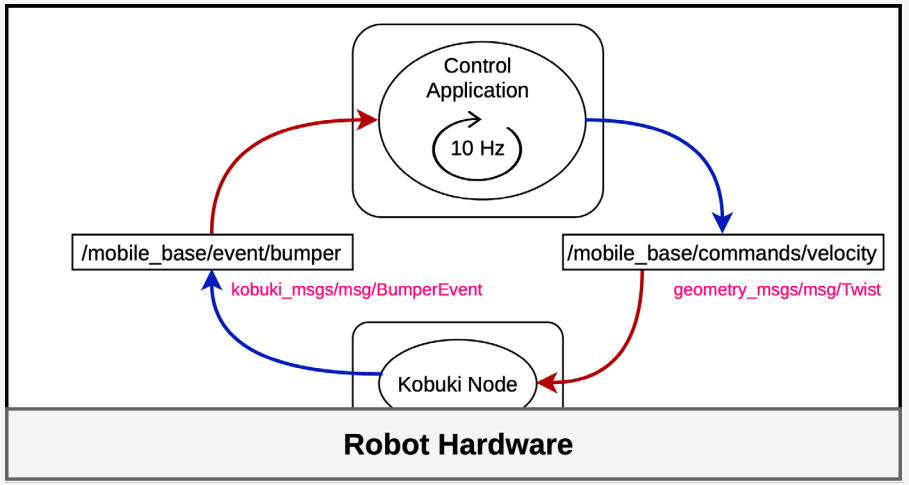
- 2017 First ROS 2 release (Ardent Apalone)
- Lot of tutorials and documentation
- We will use Ubuntu 24.04 LTS + Jazzy Jalisco LTS + Gazebo Harmonic LTS

ROS 2

The ROS 2 Computation Graph:

- Shows how a robot's software is organized and runs at runtime
- Composed of ROS 2 nodes handling tasks: sensing, control, or decision-making
- Nodes communicate via topics, services, and actions
- Forms a dynamic and distributed network
- Nodes can appear or disappear, making the system modular and flexible
- Communication mechanisms:
  - **Publication/Subscription**: Asynchronous N:M
  - **Services**: Synchronous 1:1
  - **Actions**: Asynchronous 1:1



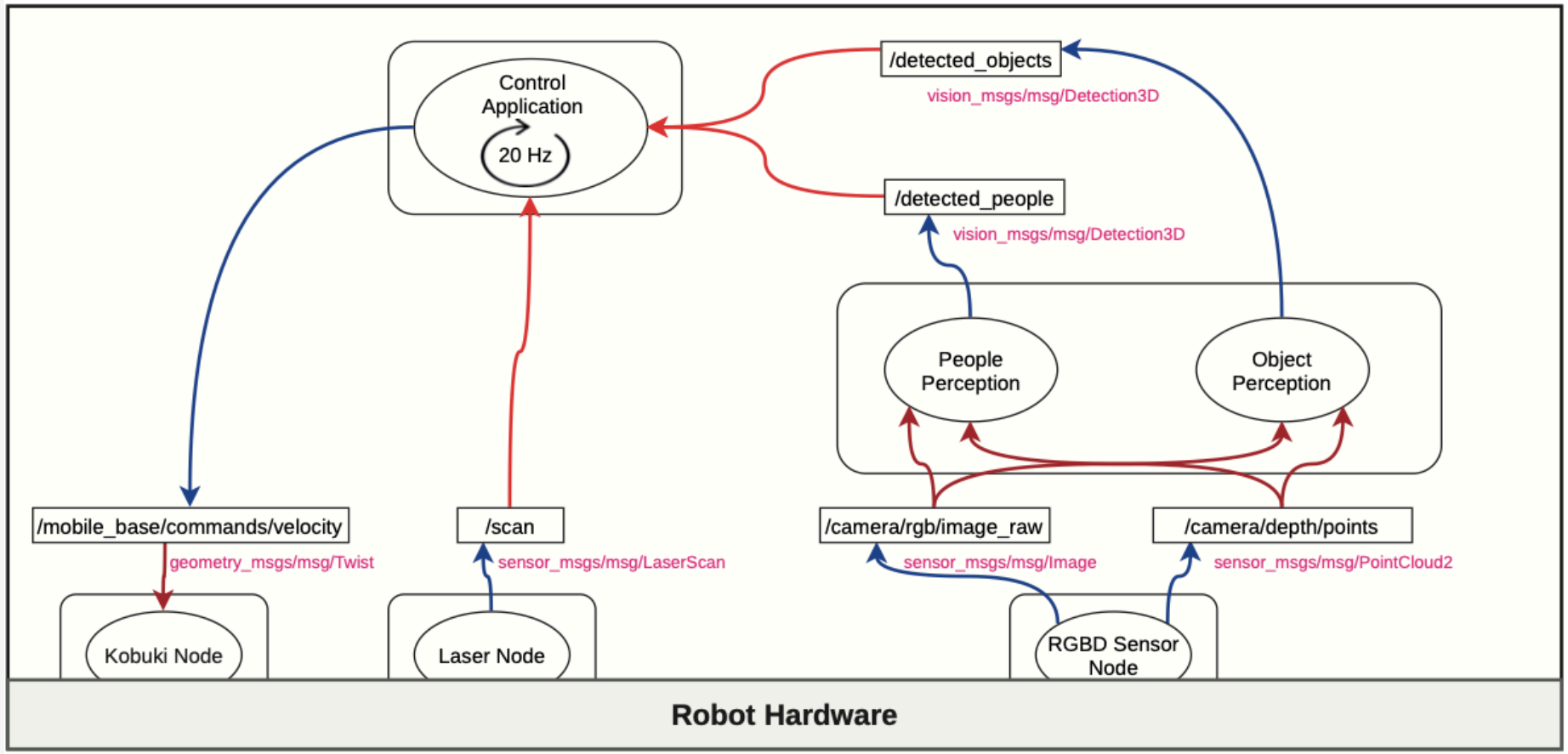


About names:

- Resources in ROS 2 follow a name convention
- The name of a resource depends on:
  - The type on name: relative, absolute or private
  - The node name
  - The namespace

name	Result: (node: my_node / ns: none)	Result: (node: my_node / ns: my_ns)
my_topic	/my_topic	/my_ns/my_topic
/my_topic	/my_topic	/my_topic
~my_topic	/my_node/my_topic	/my_ns/my_node/my_topic

Example:


Control Application
20 Hz
/detected_objects
vision_msgs/msg/Detection3D
/detected_people
vision_msgs/msg/Detection3D
People Perception
Object Perception
/mobile_base/commands/velocity
geometry_msgs/msg/Twist
/scan
sensor_msgs/msg/LaserScan
/camera/rgb/image_raw
sensor_msgs/msg/Image
/camera/depth/points
sensor_msgs/msg/PointCloud2
Kobuki Node
Laser Node
RGBD Sensor Node
Robot Hardware

The workspace

- Approaches ROS 2 software from a static point of view
- Defines where ROS 2 software is installed and organized
- Includes tools to build, manage, and launch the computation graph
- Encompasses the build system, package structure, and node startup tools
- Key Elements
  - Workspace:
    - Directory that contains one or more packages
    - Must be built and sourced to be usable
    - Supports underlays and overlays for layered developments
  - Packages:
    - Minimum functional unit of ROS 2 software
    - Contains executables, libraries, configs, or message definitions for a common purpose

The workspace

- The workspace is a folder that contains packages (e.g., named `my_ws`)
- These packages are under `src` folder:

```
my_ws/                          ← Workspace root
├── src/                        ← Source folder with ROS 2 packages
```

- To build the workspace, we must activate ROS (underlay) and compile:

```
$ cd ~/my_ws
$ source /opt/ros/jazzy/setup.bash
$ colcon build --symlink-install
```

- After building, these folders are created:

```
my_ws/                          ← Workspace root
├── src/                        ← Source folder with ROS 2 packages
├── install/                    ← Installed build outputs (after `colcon build`)
├── build/                      ← Intermediate build files
└── log/                        ← Build and run logs
```

The workspace

- After build the workspace, we must activate it (overlay):

```
$ source ~/my_ws/install/setup.bash
```

- This command makes your packages available to ROS 2 tools and terminals
- You must do this every time you open a new terminal or add it to your `.bashrc`
- Now, you can run your packages directly or using a launch file:

```
$ ros2 run my_robot_pkg my_node
$ ros2 launch my_robot_pkg my_launch_file.launch.py
```

The workspace

- Finally, a typical ROS 2 workspace looks like this after building:

```
my_ws/                      ← Workspace root
├── src/                    ← Source folder with ROS 2 packages
│   ├── my_robot_pkg/       ← Example package
│   │   ├── CMakeLists.txt  ← Build instructions
│   │   ├── package.xml     ← Package metadata
│   │   └── src/            ← Source code (nodes, libraries)
│   └── another_pkg/
├── install/                ← Installed build outputs (after `colcon build`)
│   └── setup.bash          ← Script to source the workspace (created after build)
├── build/                  ← Intermediate build files
└── log/                    ← Build and run logs
```

- Layers:
  - Underlay: The base workspace (e.g., system-installed ROS 2, or another built workspace)
  - Overlay: A custom workspace on top, which can override or extend the underlay
  - When sourced (`source install/setup.bash`), ROS 2 searches the overlay first for packages and falls back to the underlay if not found

Packages

- A package is the smallest buildable and reusable unit in ROS 2
- It groups together code, nodes, libraries, config files, and message definitions for a specific purpose:

```
my_robot_pkg/
├── package.xml          ← Package metadata (name, version, dependencies)
├── CMakeLists.txt       ← Build instructions (CMake)
├── src/                 ← Source code (nodes, libraries)
├── include/             ← Header files (if needed)
├── launch/              ← Launch files (optional)
├── config/              ← Configuration files (optional)
└── msg/ or srv/         ← Custom messages/services (optional)
```

Packages

- package.xml file: Metadata & Dependencies
  - Package name, version, description, maintainer
  - Build and run dependencies
  - License and export info

```xml
<package format="3">
  <name>my_robot_pkg</name>
  <version>0.1.0</version>
  <description>My robot's control package</description>
  <maintainer email="user@example.com">Your Name</maintainer>
  <license>Apache-2.0</license>

  <buildtool_depend>ament_cmake</buildtool_depend>
  <depend>rclcpp</depend>
  <depend>std_msgs</depend>
</package>
```

Packages

- CMakeLists.txt file: Build logic
  - The build system (ament_cmake, ament_python, etc.)
  - What to compile, include, and install
  - Dependencies and targets

```
cmake_minimum_required(VERSION 3.5)
project(my_robot_pkg)

find_package(ament_cmake REQUIRED)
find_package(rclcpp REQUIRED)
find_package(std_msgs REQUIRED)

add_executable(my_node src/my_node.cpp)
ament_target_dependencies(my_node rclcpp std_msgs)

install(TARGETS my_node DESTINATION lib/${PROJECT_NAME})

ament_package()
```

Hands on!

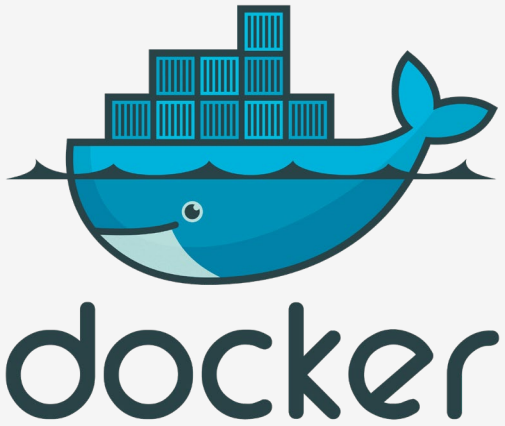
How to install ROS 2

- From binary packages
- From source

https://docs.ros.org/en/jazzy/Installation.html

Today

- Follow repo steps:
  - https://github.com/IntelligentRoboticsLabs/docker_infrastructure
- Linux users:
  - Local (Ubuntu 24.04): install ROS 2 and run script
  - Docker: build the docker or download it
- Other users:
  - Docker: download it



Docker environment (Linux)

- Run the prepared Docker (`./run_docker`) container for the course, which includes:
  - ROS 2, simulation tools, pre-configured workspace

```
$ docker run -p 6080:80 --privileged --name school -d jmguerreroh/school:ubuntu24
```

- Check the container status (`docker container ls`):
  - When the status appears as healthy, it means the ROS 2 environment is running correctly inside Docker

```
CREATED          STATUS                                PORTS
10 seconds ago   Up 10 seconds (health: starting)      0.0.0.0:6080->80/tcp, :::6080->80/tcp

CREATED              STATUS                            PORTS
About a minute ago   Up About a minute (healthy)       0.0.0.0:6080->80/tcp, :::6080->80/tcp
```

- Once the container is healthy:
  - Open your browser and go to: http://localhost:6080

Docker environment

- To stop the docker:

```
$ docker stop school
```

- To run the docker again:

```
$ docker start school
```

Docker environment

- The container includes a ROS 2 workspace: `~/ros2_ws`
- This `ws` contains all the packages needed to launch the Mirte robot simulation
- The simulation is provided by the `mirte_simulator` package
- To launch the environment, run:

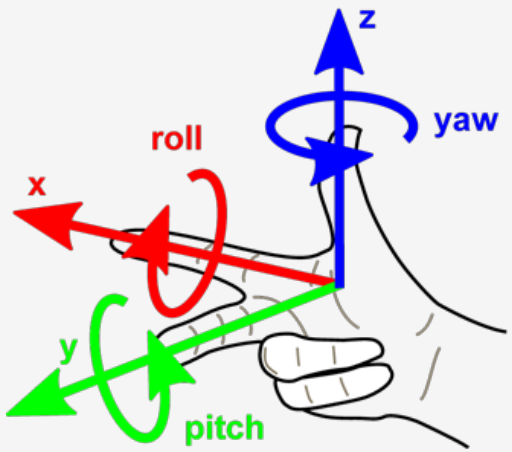
```
$ ros2 launch mirte_gazebo mirte_simulation.launch.py
```

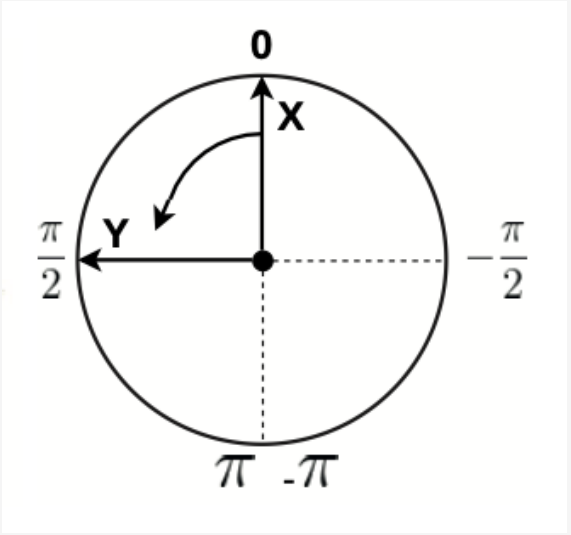
- This command starts the Mirte robot simulation, including the robot model, world, and controllers

Docker environment

- Move the robot:

```
$ ros2 topic pub /cmd_vel geometry_msgs/msg/Twist "{linear:
{x: 1.0, y: 0.0, z: 0.0}, angular: {x: 0.0, y: 0.0, z: 0.0}}"
```





First steps in Terminal

```
$ ros2

usage: ros2 [-h] Call ‘ros2 <command> -h‘ for more detailed usage. ...
ros2 is an extensible command-line tool for ROS 2.

...
```

```
ros2 <command> <verb> [<params>|<option>]*
```

```
action      extension_points  node        test
bag         extensions        param       topic
component   interface         pkg         wtf
launch      run               daemon      lifecycle
security    doctor            multicast   service
```

Further readings:

- https://github.com/ros2/ros2cli
- https://github.com/ubuntu-robotics/ros2_cheats_sheet/blob/master/cli/cli_cheats_sheet.pdf

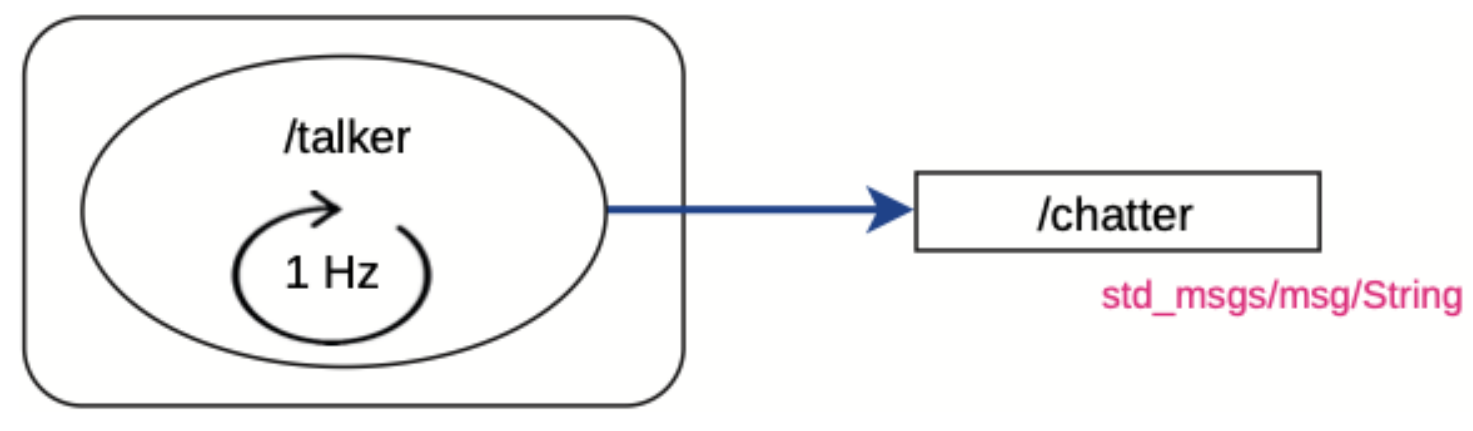
Packages

```
$ ros2 pkg list

ackermann_msgs
action_msgs
action_tutorials_cpp
...
```

```
$ ros2 pkg executables demo_nodes_cpp

demo_nodes_cpp add_two_ints_client
demo_nodes_cpp add_two_ints_client_async
demo_nodes_cpp add_two_ints_server
demo_nodes_cpp allocator_tutorial
...
demo_nodes_cpp talker
...
```

Running a ROS 2 program

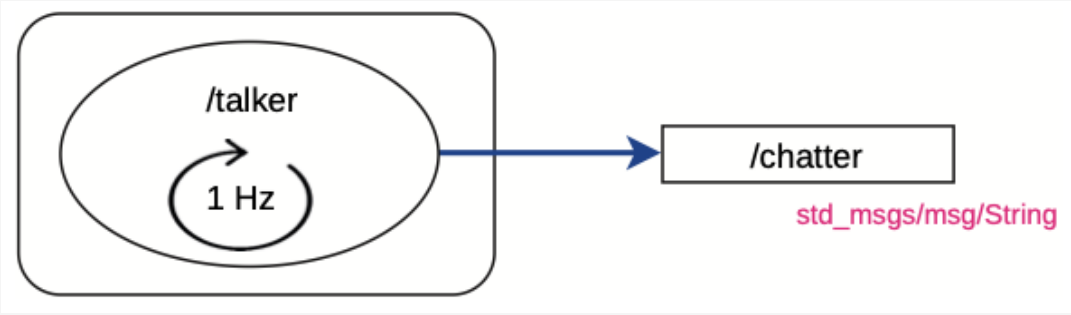
```
$ ros2 run demo_nodes_cpp talker

[INFO] [1643218362.316869744] [talker]: Publishing: 'Hello World: 1'
[INFO] [1643218363.316915225] [talker]: Publishing: 'Hello World: 2'
[INFO] [1643218364.316907053] [talker]: Publishing: 'Hello World: 3'
...
```



Running a ROS 2 program

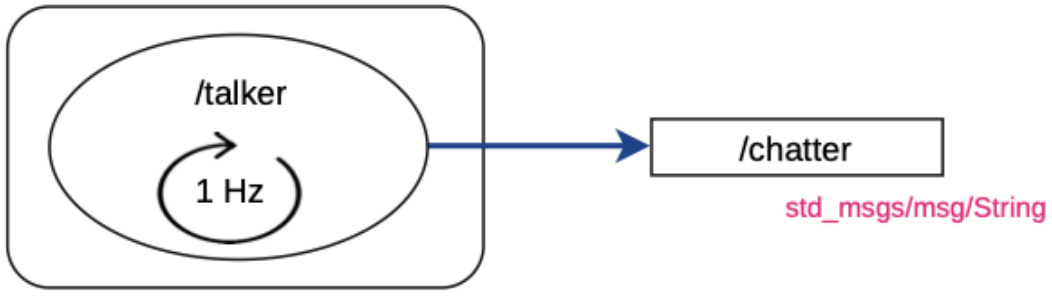
```
$ ros2 node list

/talker
```

```
$ ros2 topic list

/chatter
/parameter_events
/rosout
```



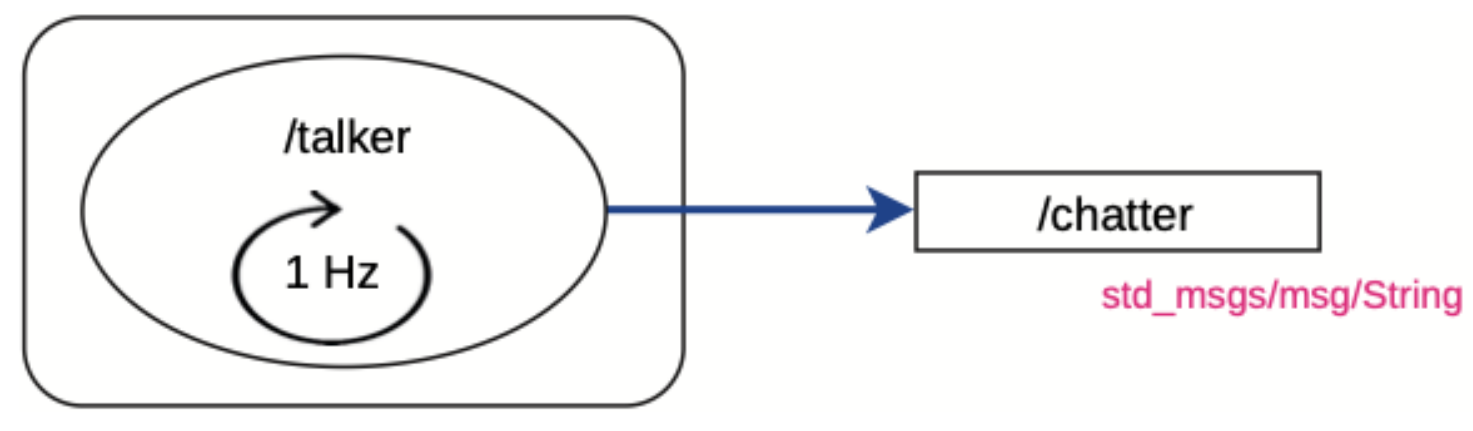
Running a ROS 2 program

```
$ ros2 node info /talker

/talker
   Subscribers:
       /parameter_events: rcl_interfaces/msg/ParameterEvent
   Publishers:
       /chatter: std_msgs/msg/String
       /parameter_events: rcl_interfaces/msg/ParameterEvent
       /rosout: rcl_interfaces/msg/Log
   Service Servers:

...
```



Running a ROS 2 program

```
$ ros2 topic info /chatter

Type: std_msgs/msg/String
Publisher count: 1

Subscription count: 0
```



Interfaces

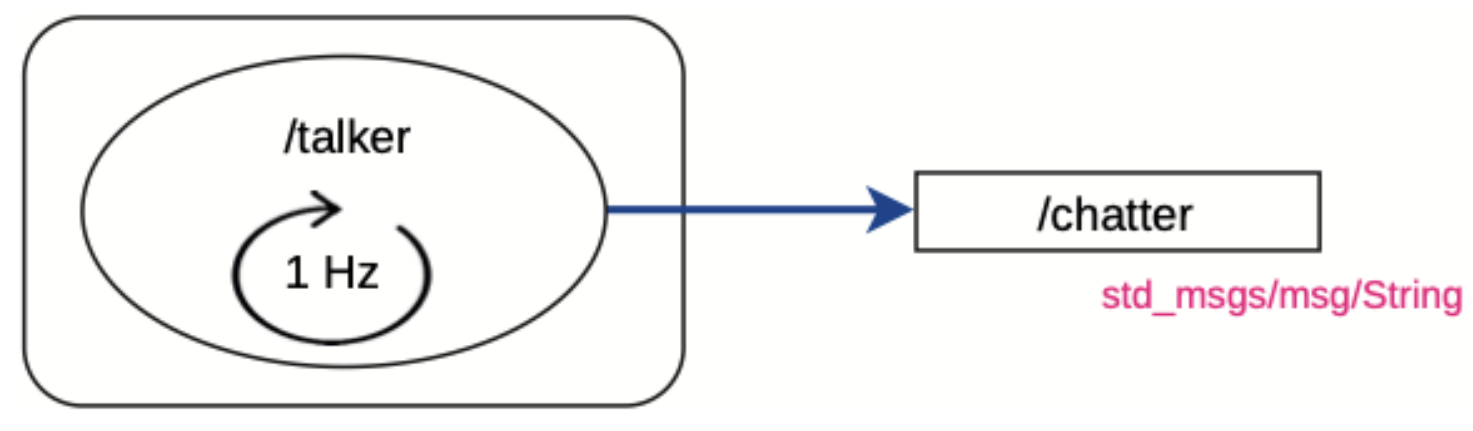
- Define the data structure and communication types used between nodes:
  - **Messages** (.msg) for topic-based data exchange
  - **Services** (.srv) for request-response patterns
  - **Actions** (.action) for long-running goals with feedback and result

```
$ ros2 interface list

Messages:
    ackermann_msgs/msg/AckermannDrive
    ackermann_msgs/msg/AckermannDriveStamped
    ...
    visualization_msgs/msg/MenuEntry
Services:
    action_msgs/srv/CancelGoal
    ...
    visualization_msgs/srv/GetInteractiveMarkers
Actions:
    action_tutorials_interfaces/action/Fibonacci
    ...
```

```
$ ros2 interface show std_msgs/msg/String

... comments
string data
```

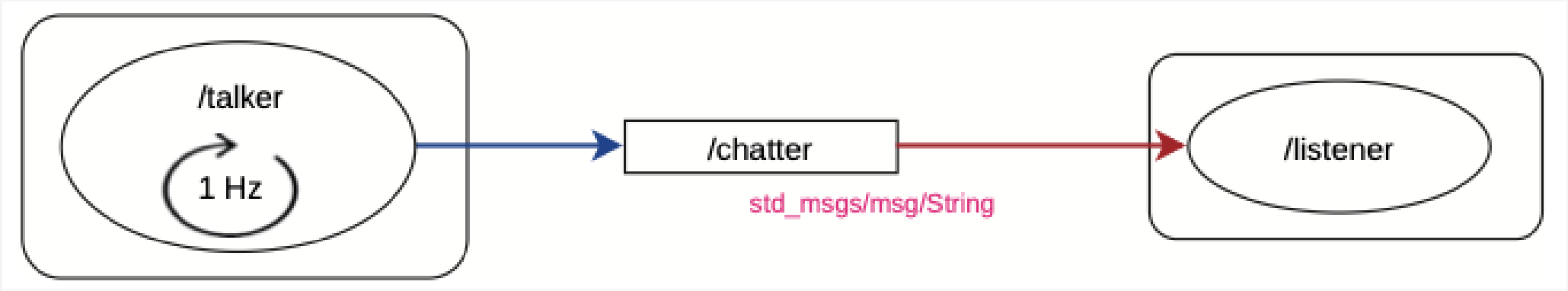
Inspect a topic

```
$ ros2 topic echo /chatter

data: 'Hello World: 1578'
---
data: 'Hello World: 1579'
...
```



Running a listener

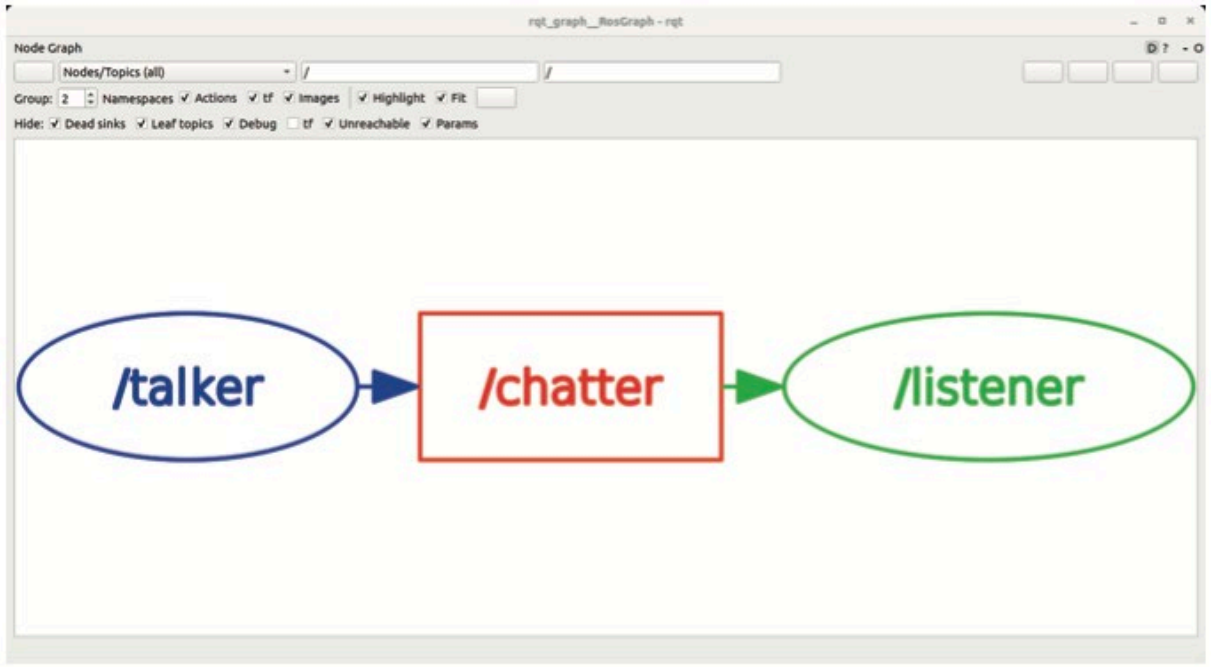
```
$ ros2 run demo_nodes_py listener

[INFO] [1643220136.232617223] [listener]: I heard: [Hello World: 1670]
[INFO] [1643220137.197551366] [listener]: I heard: [Hello World: 1671]
[INFO] [1643220138.198640098] [listener]: I heard: [Hello World: 1672]
...
```



RQT tools

- GUI-based utilities to visualize, monitor, and debug topics, nodes, parameters…

```
$ ros2 run rqt_graph rqt_graph
```



RViz2

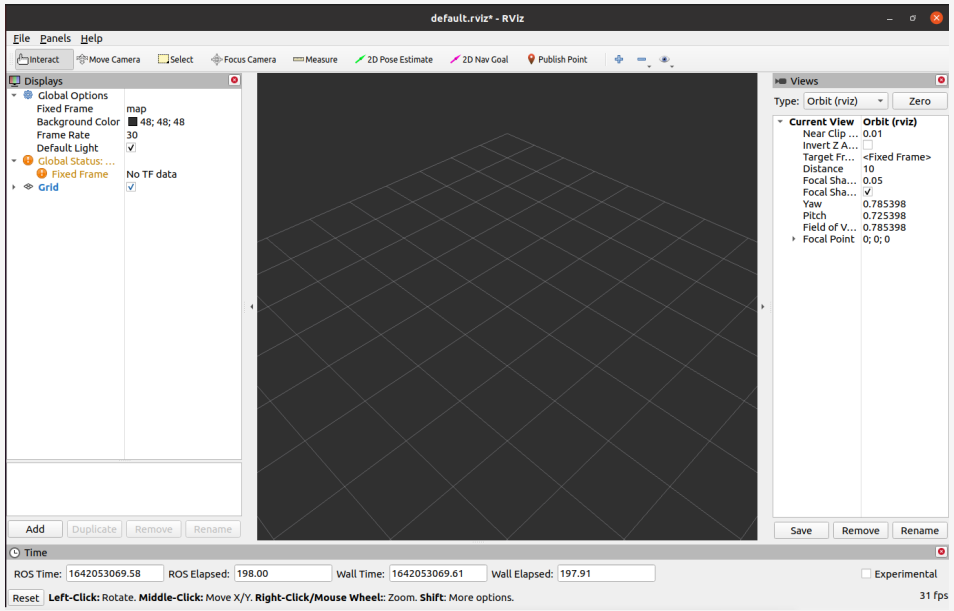
- Open a new terminal and enter the command to open RViz2

```
$ rviz2
```

* Sometimes is necessary to source the workspace to access different messages and paths:

```
$ source ~/ros2_ws/install/setup.bash
```

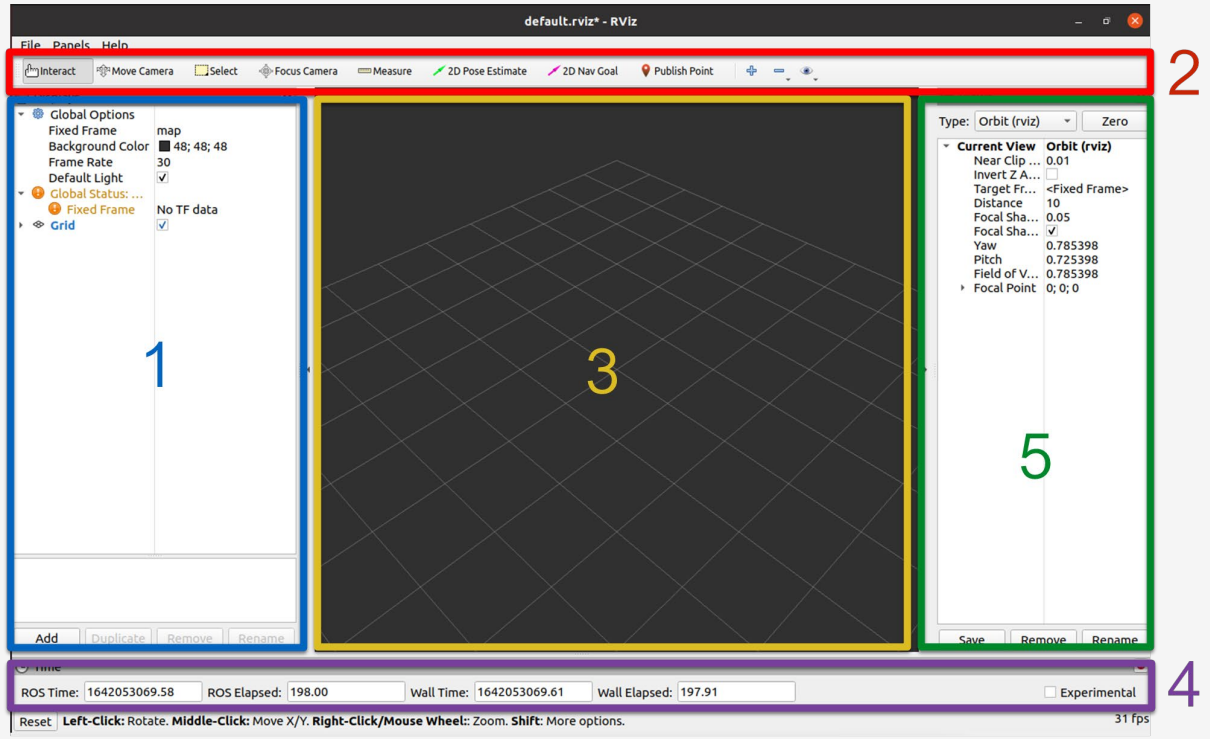
- The next window will be open:



RViz2: Interface overview

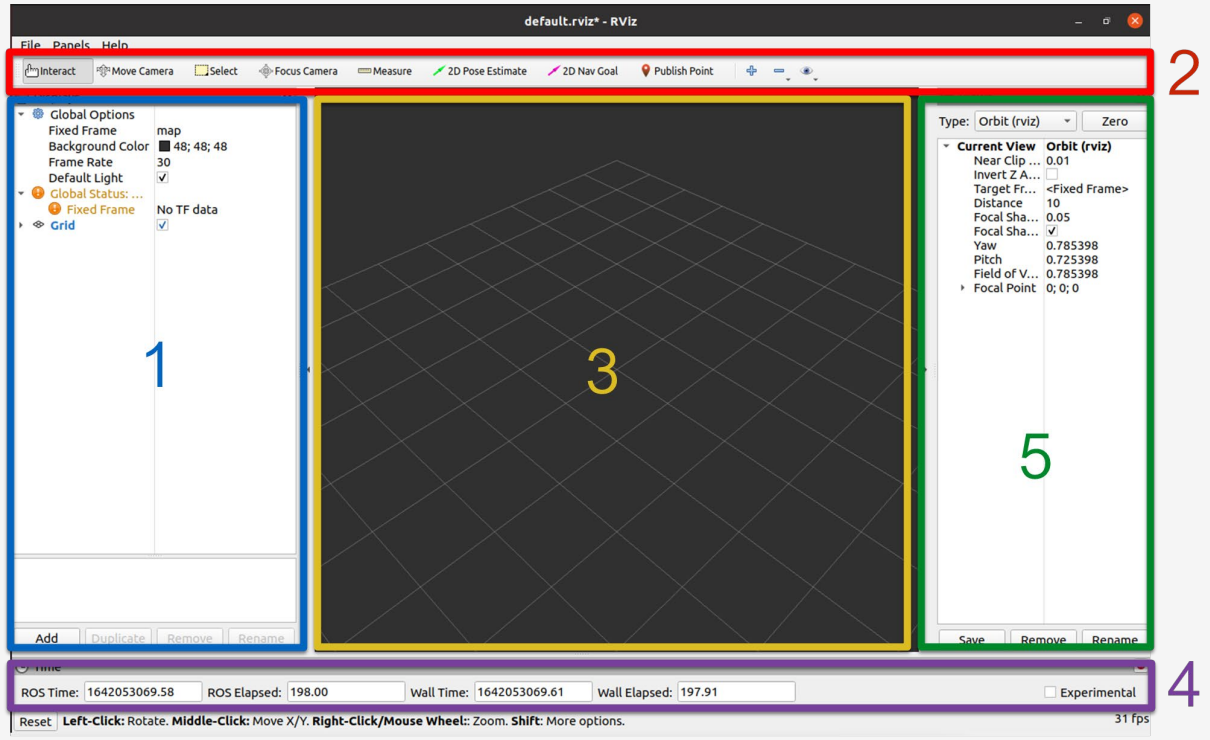
1. Monitors
   - Visual elements that render data in the 3D world
   - Each monitor may include adjustable options in the Display panel

2. Toolbar
   - Provides quick access to tools (e.g., Move Camera, Select, Measure)
   - Supports interaction with the scene or loaded data



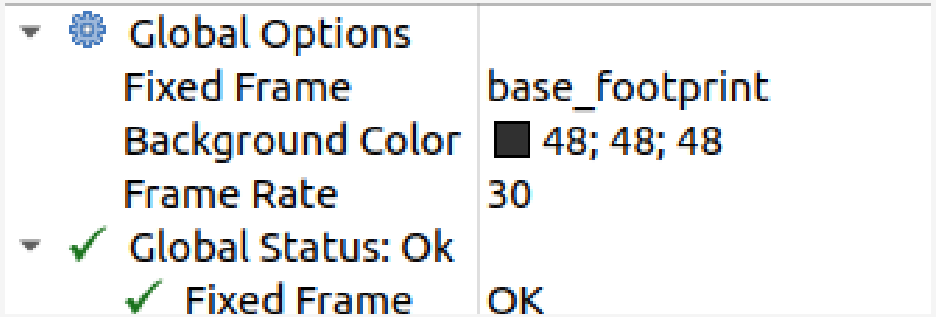
RViz2: Interface overview

3. 3D View (Central Panel)
   - Main visualization area
   - Displays sensor data, robot models, environments, and paths in 3D
4. Time display panel
   - Shows System Time and ROS Time
   - Useful for simulation vs real-time debugging
5. View control panel
   - Allows users to change observation angles
   - Includes pre-set views (Top, Side, Orbit, etc.)



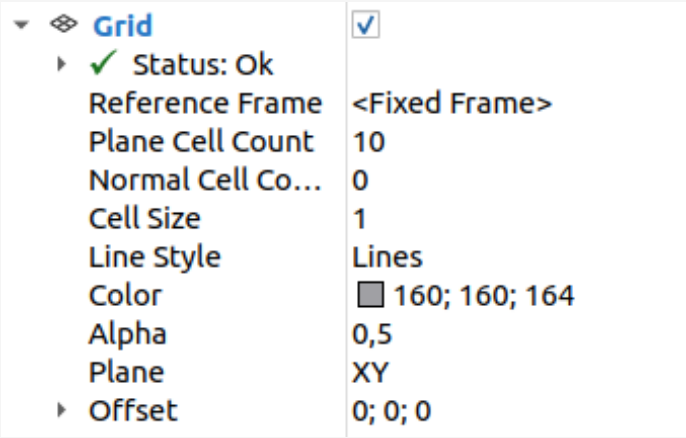
RViz2: Global options

- Key Parameters:
  - Fixed Frame
    - The reference coordinate frame for all visualized data
    - Chosen from available TF frames (combo box)
    - Common choices: `map` or `odom`
  - Frame Rate
    - Controls how often the 3D view is refreshed
    - Recommended values: 30 or 60 FPS for smooth rendering



RViz2: Grid plugin

- Used to visualize a reference grid on the floor or other planes, helping to orient objects in 3D space
- Key Parameters:
  - Reference Frame: Coordinate frame for the grid (usually the Fixed Frame)
  - Plane Cell Count: Number of grid cells along the plane axes (width and depth)
  - Normal Cell Count: Cells along the axis perpendicular to the grid plane (0 for flat floors)
  - Cell Size: Dimensions of each grid cell in meters
  - Plane: Axes defining the grid's plane (e.g., XY, XZ, YZ)

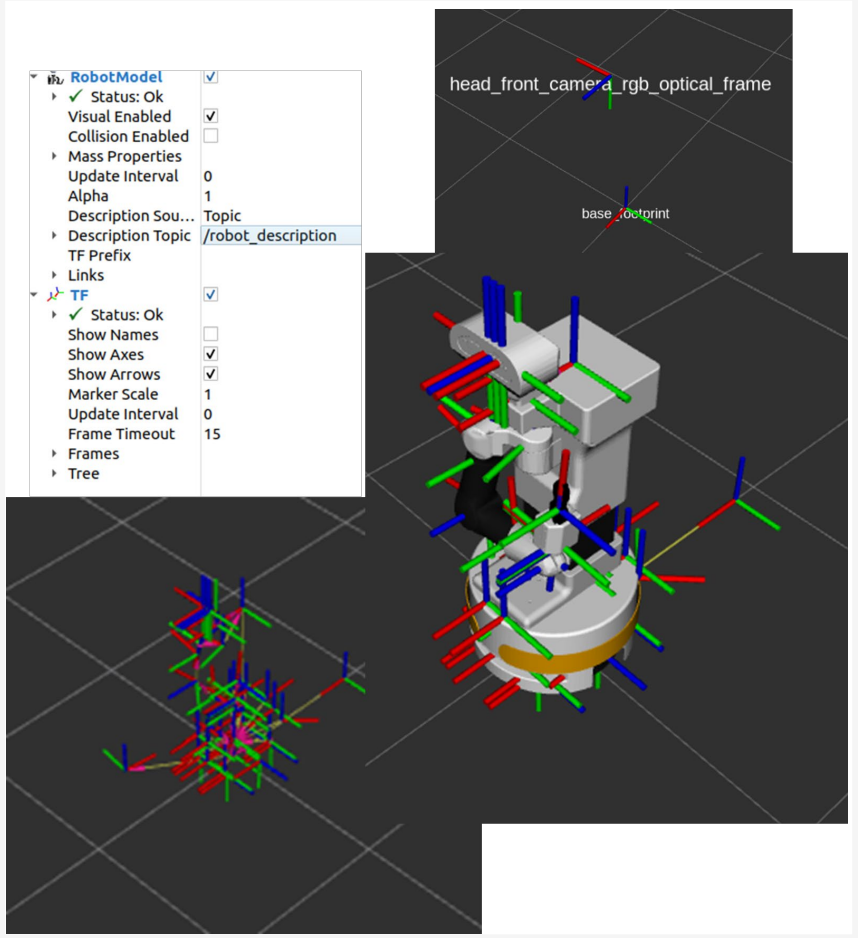


RViz2: Robot model plugin

- Displays the robot’s 3D model based on its URDF
- Useful for checking link/joint positions and transformations in real time
- Key Parameters:)
  - Visual Enabled: Show/hide the robot's 3D model
  - Description Source: load the model from:
    - File: Local URDF file
    - Topic: ROS topic (e.g., /robot_description)
  - Links Tree: View link/joint hierarchy relative to Fixed Frame

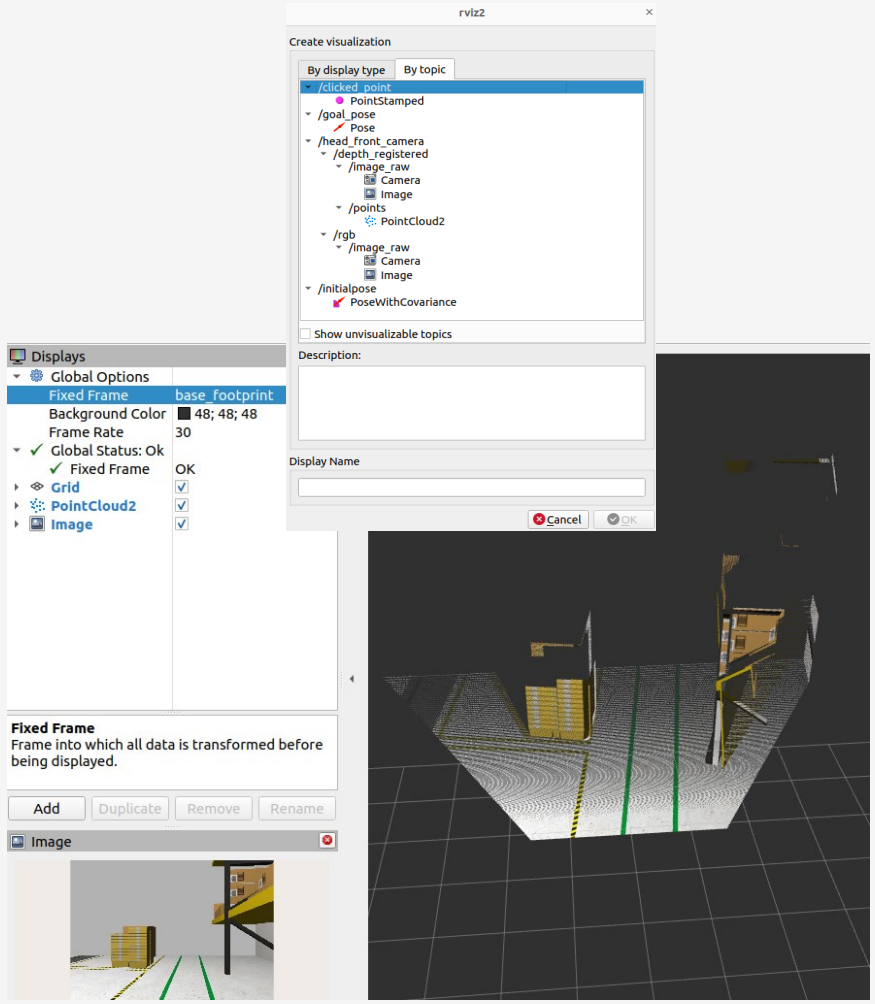
RViz2: TF plugin

- Visualizes the TF tree (positions & orientations of frames), critical for understanding the robot's structure in real time
- Key Parameters:
  - Show Names – Display frame names in 3D
  - Show Axes – Show X/Y/Z orientation axes
  - Show Arrows – Visualize frame connections
  - Marker Scale – Resize TF visuals for clarity
  - Update Interval – Set refresh rate (0 = always)
- Toggle individual frames to reduce clutter and focus on what matters



RViz2: Adding displays

- By default, Global Options and Grid are loaded
- To add more displays, click the “Add” button at the bottom of the Displays panel
- You can also add displays directly by selecting topics from the By topic tab
- Common additions include:
  - Camera
  - PointCloud2
  - Marker / MarkerArray (for visual markers)
- Use displays to visualize real-time data from your robot’s sensors and systems



Developing time!

Package creation

- You can create manually or using `pkg create`

```
$ cd ~/my_ws/src
$ ros2 pkg create --build-type ament_cmake --dependencies rclcpp std_msgs \
      --node-name my_node my_robot_pkg
```

```
my_ws/                          ← Workspace root
└── src/                        ← Source folder with ROS 2 packages
    └── my_robot_pkg/           ← Example package
        ├── CMakeLists.txt      ← Build instructions
        ├── package.xml         ← Package metadata
        ├── include             ← Header files (if needed)
        │   └── my_robot_pkg/   ← Header files for the package
        └── src/                ← Source code (nodes, libraries)
            └── my_node.cpp     ← Example node source code
```

Files

- package.xml and CMakeLists.txt

```xml
<package format="3">
  <name>my_robot_pkg</name>
  <version>0.0.0</version>
  <description>TODO: Package description</description>
  <maintainer email="josemiguel.guerrero@urjc.es">jmguerrero</maintainer>
  <license>TODO: License declaration</license>

  <buildtool_depend>ament_cmake</buildtool_depend>

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

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

  <export>
    <build_type>ament_cmake</build_type>
  </export>
</package>
```

```
cmake_minimum_required(VERSION 3.8)
project(my_robot_pkg)

if(CMAKE_COMPILER_IS_GNUCXX OR CMAKE_CXX_COMPILER_ID MATCHES "Clang")
  add_compile_options(-Wall -Wextra -Wpedantic)
endif()

# find dependencies
find_package(ament_cmake REQUIRED)
find_package(rclcpp REQUIRED)
find_package(std_msgs REQUIRED)

add_executable(my_node src/my_node.cpp)
target_include_directories(my_node PUBLIC
  $<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}/include>
  $<INSTALL_INTERFACE:include/${PROJECT_NAME}>)
target_compile_features(my_node PUBLIC c_std_99 cxx_std_17)  # Require C99 and C++17
ament_target_dependencies(
  my_node
  "rclcpp"
  "std_msgs"
)

install(TARGETS my_node
  DESTINATION lib/${PROJECT_NAME})

if(BUILD_TESTING)
  find_package(ament_lint_auto REQUIRED)
  # the following line skips the linter which checks for copyrights
  # comment the line when a copyright and license is added to all source files
  set(ament_cmake_copyright_FOUND TRUE)
  # the following line skips cpplint (only works in a git repo)
  # comment the line when this package is in a git repo and when
  # a copyright and license is added to all source files
  set(ament_cmake_cpplint_FOUND TRUE)
  ament_lint_auto_find_test_dependencies()
endif()

ament_package()
```

Make a node

- `Class MyNode`
- Inherit from `rclcpp::Node`
- `RCLCPP_*` to show messages
- `main` function
  - Create nodes
  - Spin nodes

```cpp
#include <rclcpp/rclcpp.hpp>
#include <std_msgs/msg/string.hpp>

class MyNode : public rclcpp::Node
{
public:
  MyNode()
  : Node("my_node")
  {
    RCLCPP_INFO(this->get_logger(), "Node %s has been created", this->get_name());
  }
  ~MyNode()
  {
    RCLCPP_INFO(this->get_logger(), "Node %s is being destroyed", this->get_name());
  }
};

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

Spin nodes and callbacks

- ROS 2 nodes must be spun to process incoming data
- rclcpp offers three spinning options depending on how you want to handle callbacks
- A callback is a function automatically executed in response to an event, like receiving a message or a timer expiring
- In ROS 2, callbacks are often used in subscriptions, services, and timers

```
rclcpp::TimerBase::SharedPtr timer_;
timer_ = create_wall_timer(
  500ms, std::bind(&BumpGoNode::step, this));
```

```
rclcpp::Subscription<std_msgs::msg::Int32>::SharedPtr subscription_;

subscription_ = this->create_subscription<std_msgs::msg::Int32>(
  "/counter", 10, std::bind(&SubscriberNode::callback, this, _1));
```

Spin nodes and callbacks

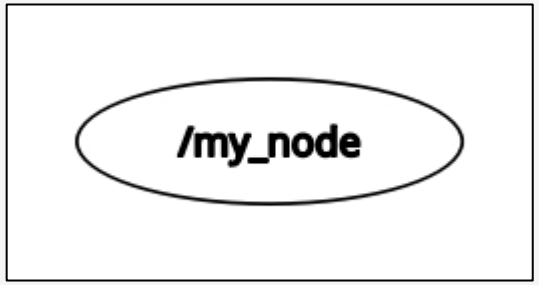
1. `rclcpp::spin(node);`
   - Spins forever (loop), blocking the thread
   - Processes all callbacks as they arrive
   - Best for standalone ROS nodes

2. `rclcpp::spin_some(node);`
   - Non-blocking, returns immediately
   - Processes only ready callbacks
   - Ideal for integration with other event loops (e.g., GUI, game loop)

3. `rclcpp::spin_once(node);`
   - Blocks briefly (default: 100ms)
   - Waits, then processes available callbacks
   - Useful for periodic polling

Build and run

```
$ colcon build --symlink-install --packages-select my_robot_pkg
$ source install/setup.bash

$ ros2 run my_robot_pkg my_node
[INFO] [1750752397.513274473] [my_node]: Node my_node has been created
^C[INFO] [1750752403.438272160] [rclcpp]: signal_handler(signum=2)
[INFO] [1750752403.439806346] [my_node]: Node my_node is being destroyed
```




```
$ ros2 node list
/my_node
```

Publisher

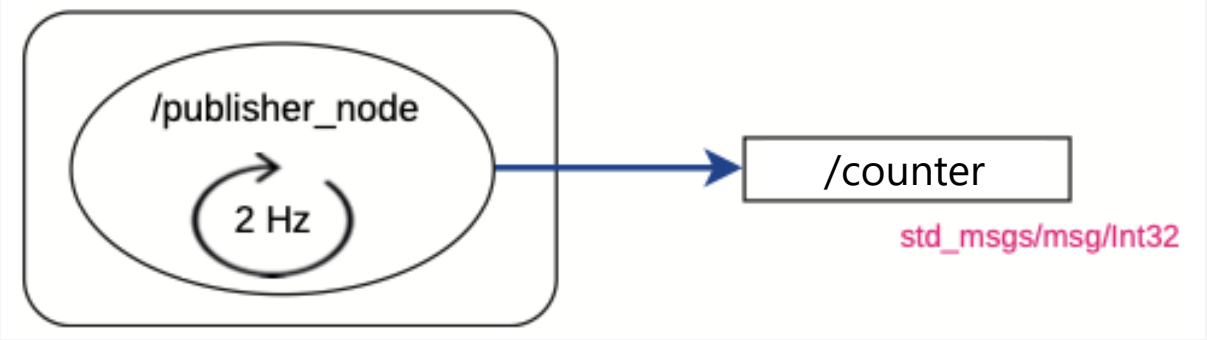
- Inherit from `rclcpp::Node` helps to organize better your code
- Control execution cycle **internally** with timers

```cpp
// For std_msgs/msg/Int32
#include "std_msgs/msg/int32.hpp"

std_msgs::msg::Int32 msg_int32;

// For sensor_msgs/msg/LaserScan
#include "sensor_msgs/msg/laser_scan.hpp"

sensor_msg::msg::LaserScan msg_laserscan;
```

```cpp
class PublisherNode : public rclcpp::Node
{
public:
  PublisherNode()
  : Node("publisher_node"), counter_(0)
  {
    publisher_ = this->create_publisher<std_msgs::msg::Int32>("/counter", 10);
    timer_ = this->create_wall_timer(500ms, std::bind(&PublisherNode::step, this));
  }

  void step()
  {
    std_msgs::msg::Int32 message;
    message.data = counter_++;
    RCLCPP_INFO(this->get_logger(), "Publishing: '%d'", message.data);
    publisher_->publish(message);
  }

private:
  rclcpp::Publisher<std_msgs::msg::Int32>::SharedPtr publisher_;
  rclcpp::TimerBase::SharedPtr timer_;
  int counter_ = 0;
};
```

Publisher

```
$ ros2 run publisher_example publisher_example

[INFO] [1750753580.683567996] [publisher_node]: Publishing: '0'
[INFO] [1750753581.183693278] [publisher_node]: Publishing: '1'
[INFO] [1750753581.683872955] [publisher_node]: Publishing: '2'
[INFO] [1750753582.183637578] [publisher_node]: Publishing: '3'
[INFO] [1750753582.683418684] [publisher_node]: Publishing: '4'
[INFO] [1750753583.183715430] [publisher_node]: Publishing: '5'
```



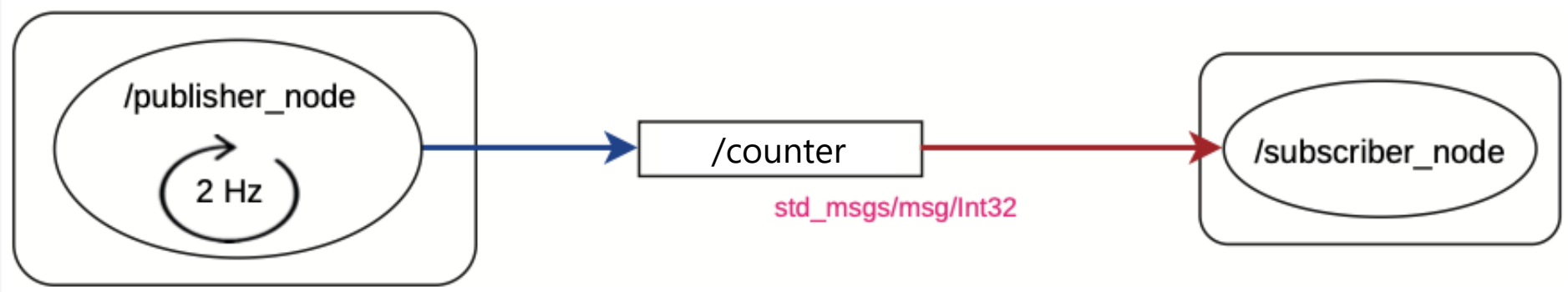
Subscriber

- Subscribes to a specific topic to receive messages published by other nodes
- Automatically triggers a callback function when a new message is received
- The callback processes the incoming data, e.g., printing it, storing it, or using it for control logic

```
class SubscriberNode : public rclcpp::Node
{
public:
  SubscriberNode()
  : Node("subscriber_node")
  {
    subscription_ = this->create_subscription<std_msgs::msg::Int32>(
      "/counter", 10, std::bind(&SubscriberNode::callback, this, _1));
  }

private:

  void callback(const std_msgs::msg::Int32::SharedPtr msg)
  {
    RCLCPP_INFO(this->get_logger(), "Received: '%d'", msg->data);
  }

  rclcpp::Subscription<std_msgs::msg::Int32>::SharedPtr subscription_;
};
```

Subscriber

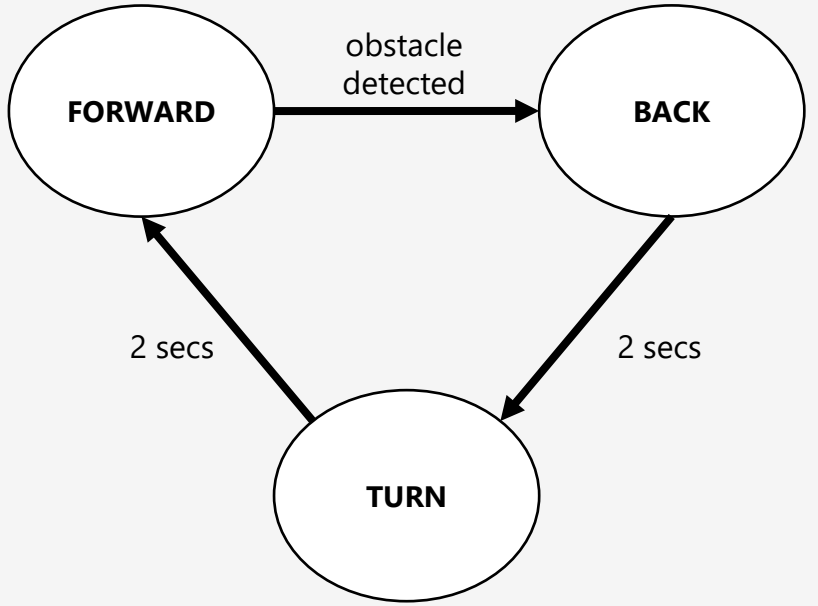
```
$ ros2 run subscriber_example subscriber_example

[INFO] [1750757636.430951420] [subscriber_node]: Received: '3871'
[INFO] [1750757636.930882816] [subscriber_node]: Received: '3872'
[INFO] [1750757637.431017626] [subscriber_node]: Received: '3873'
[INFO] [1750757637.931247113] [subscriber_node]: Received: '3874'
[INFO] [1750757638.432221658] [subscriber_node]: Received: '3875'
[INFO] [1750757638.931969091] [subscriber_node]: Received: '3876'
```



Your time is now

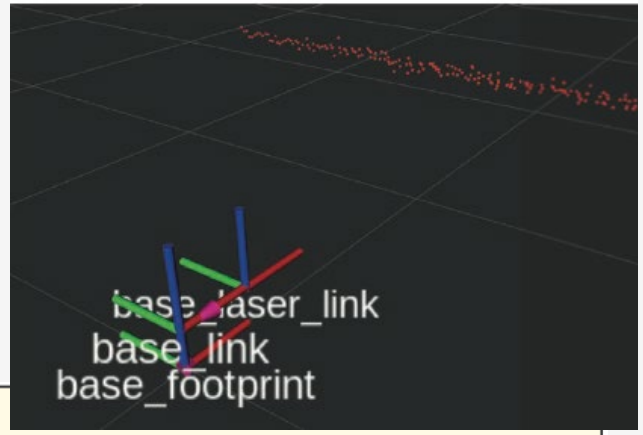
Finite State Machines (FSMs)

- Objective: Implement a reactive obstacle avoidance behavior using a Bump-and-Go strategy
- FSMs provide a simple yet effective method for encoding robot behaviors through a set of defined states and transitions between them
- Key Concepts:
  - States represent distinct modes or behaviors of the system
  - Transitions define the conditions under which the system switches from one state to another

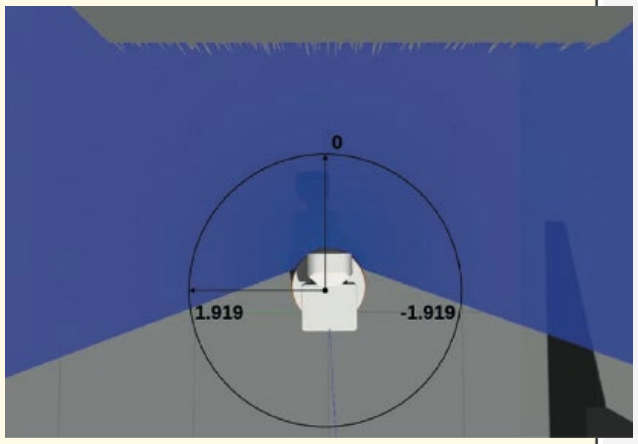


Standard interfaces: Laser sensor

```
$ ros2 interface show /sensor_msgs/msg/LaserScan

# Single scan from a planar laser range-finder
#
# If you have another ranging device with different behavior (e.g. a sonar
# array), please find or create a different message, since applications
# will make fairly laser-specific assumptions about this data

std_msgs/Header header  # timestamp in the header is the acquisition time of
                        # the first ray in the scan.
                        #
                        # in frame frame_id, angles are measured around
                        # the positive Z axis (counterclockwise, if Z is up)
                        # with zero angle being forward along the x axis

float32 angle_min       # start angle of the scan [rad]
float32 angle_max       # end angle of the scan [rad]
float32 angle_increment # angular distance between measurements [rad]

float32 time_increment  # time between measurements [seconds] - if your scanner
                        # is moving, this will be used in interpolating pos
                        # of 3d points
float32 scan_time       # time between scans [seconds]

float32 range_min       # minimum range value [m]
float32 range_max       # maximum range value [m]

float32[] ranges        # range data [m]
                        # (Note: values < range_min or > range_max should be
                        # discarded)
float32[] intensities   # intensity data [device-specific units]. If your
                        # device does not provide intensities, please leave
                        # the array empty.
```

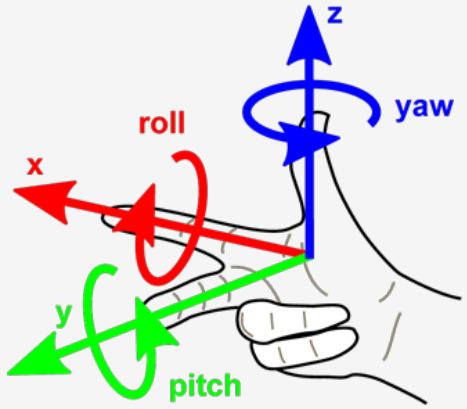



```
$ ros2 topic echo /scan_raw --no-arr

---
header:
  stamp:
    sec: 11071
    nanosec: 445000000
  frame_id: base_laser_link
angle_min: -1.9198600053787231
angle_max: 1.9198600053787231
angle_increment: 0.005774015095084906
time_increment: 0.0
scan_time: 0.0
range_min: 0.05000000074505806
range_max: 25.0
ranges: '<sequence type: float, length: 666>'
intensities: '<sequence type: float, length: 666>'
---
```



Standard interfaces: Base actuator

```
$ ros2 interface show geometry_msgs/msg/Twist

Vector3 linear
Vector3 angular

$ ros2 interface show geometry_msgs/msg/Vector3

float64 x
float64 y
float64 z
```

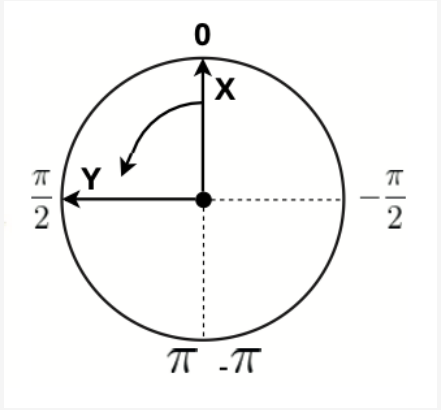


- Move forward

```
$ ros2 topic pub /cmd_vel geometry_msgs/msg/Twist "{linear:
{x: 1.0, y: 0.0, z: 0.0}, angular: {x: 0.0, y: 0.0, z: 0.0}}"
```

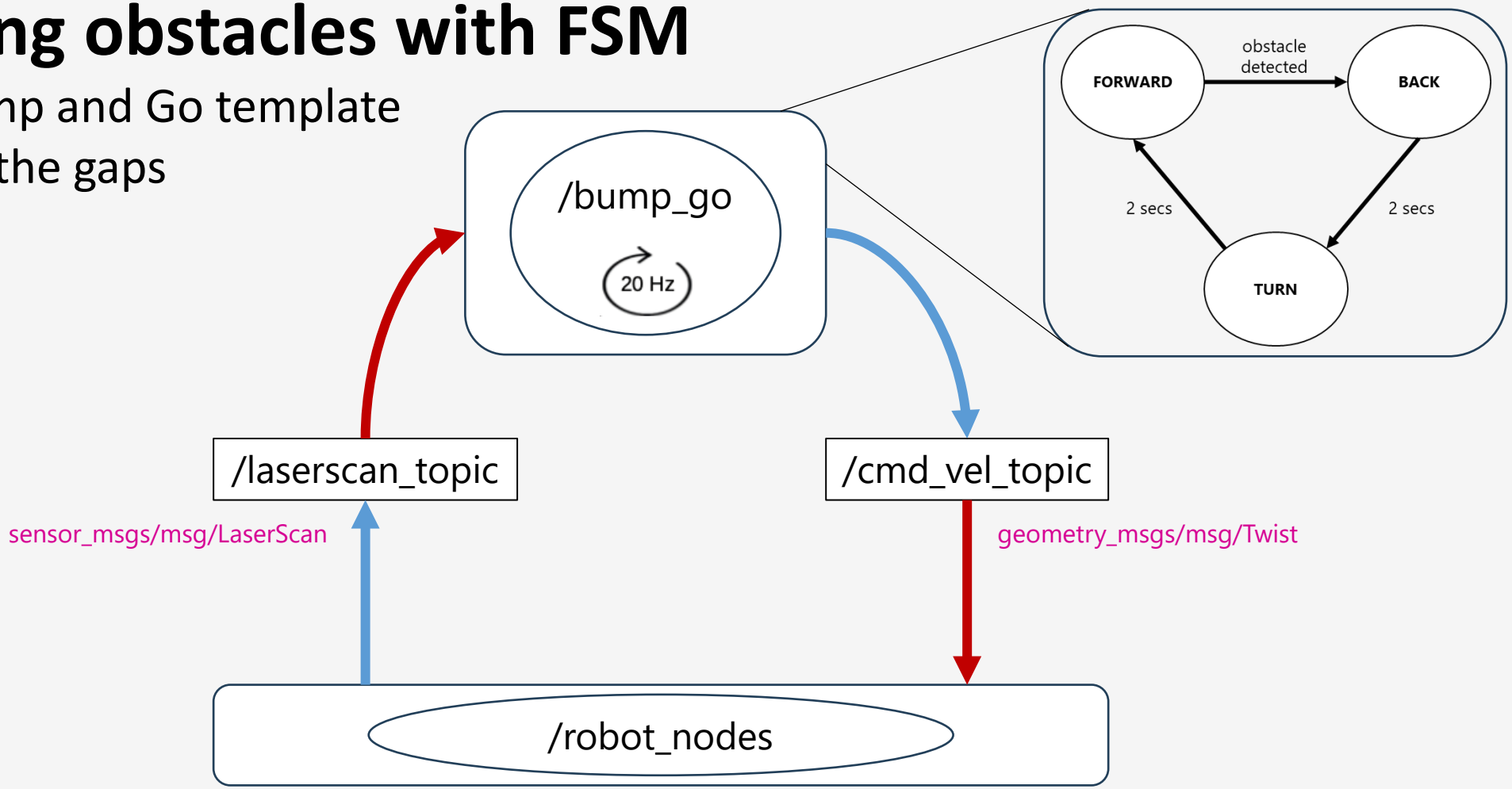
- Turn

```
$ ros2 topic pub /cmd_vel geometry_msgs/msg/Twist "{linear:
{x: 0.0, y: 0.0, z: 0.0}, angular: {x: 0.0, y: 0.0, z: 1.0}}"
```



Avoiding obstacles with FSM

- Bump and Go template
- Fill the gaps



Thank you

The CoreSense project has received funding from the European Union's Horizon Europe research and innovation programme under grant agreement No. 10107054.

License