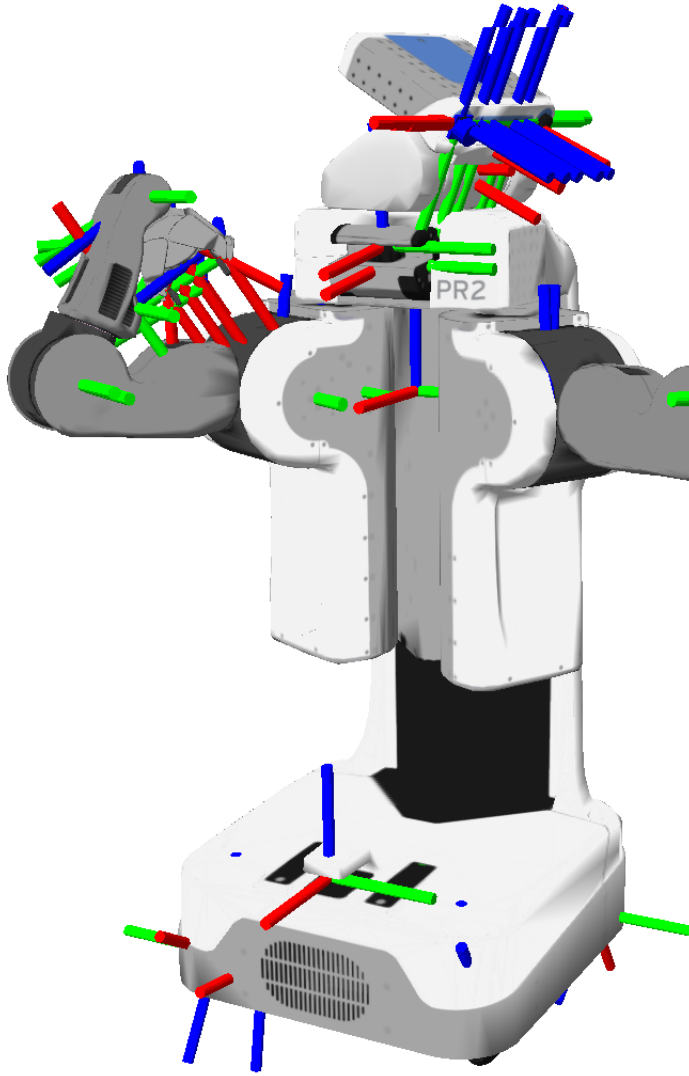
Introduction to ROS 2

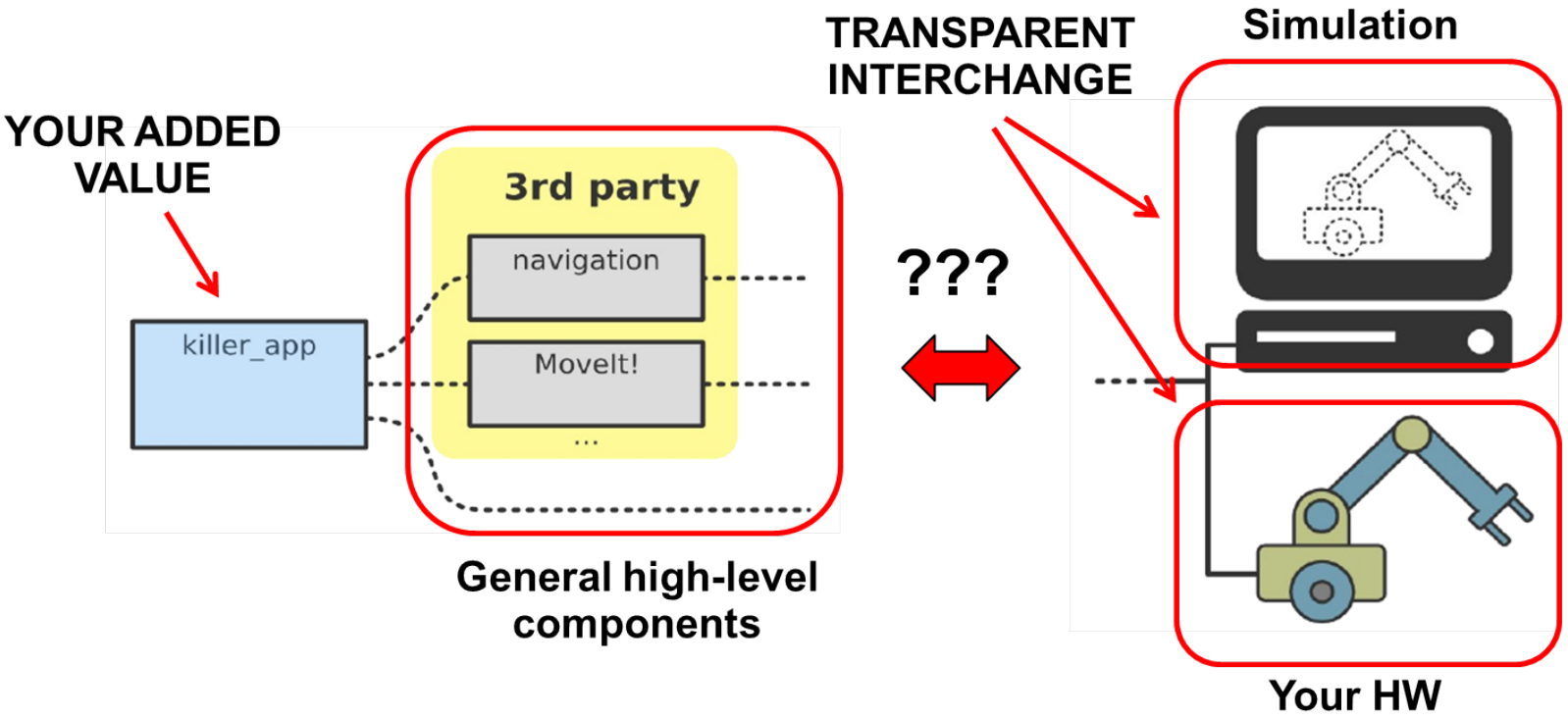
Gianluca Palli

gianluca.palli@unibo.it

DEI - LAR
University of Bologna
Viale del Risorgimento 2
40136 Bologna



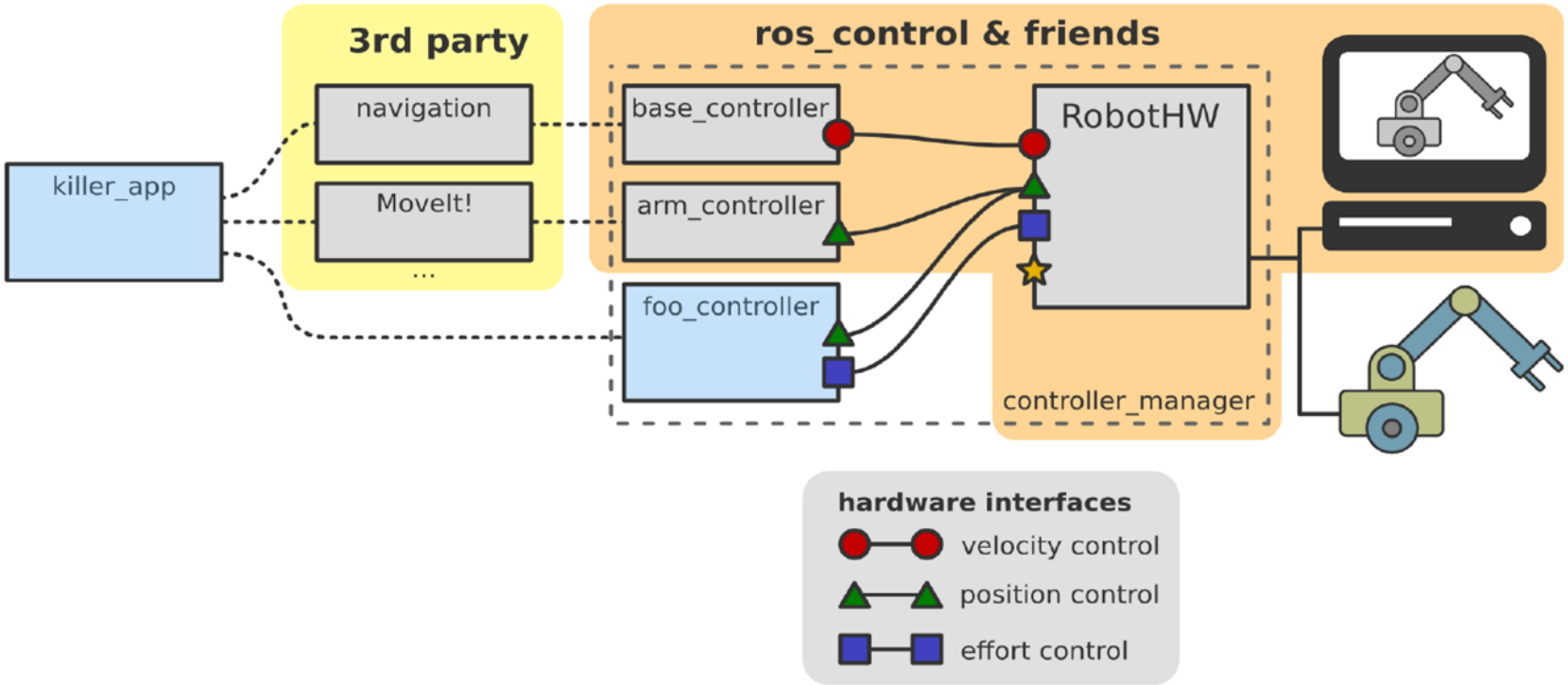
What ROS is useful for?



How to connect them?
Which interface?
Data types?

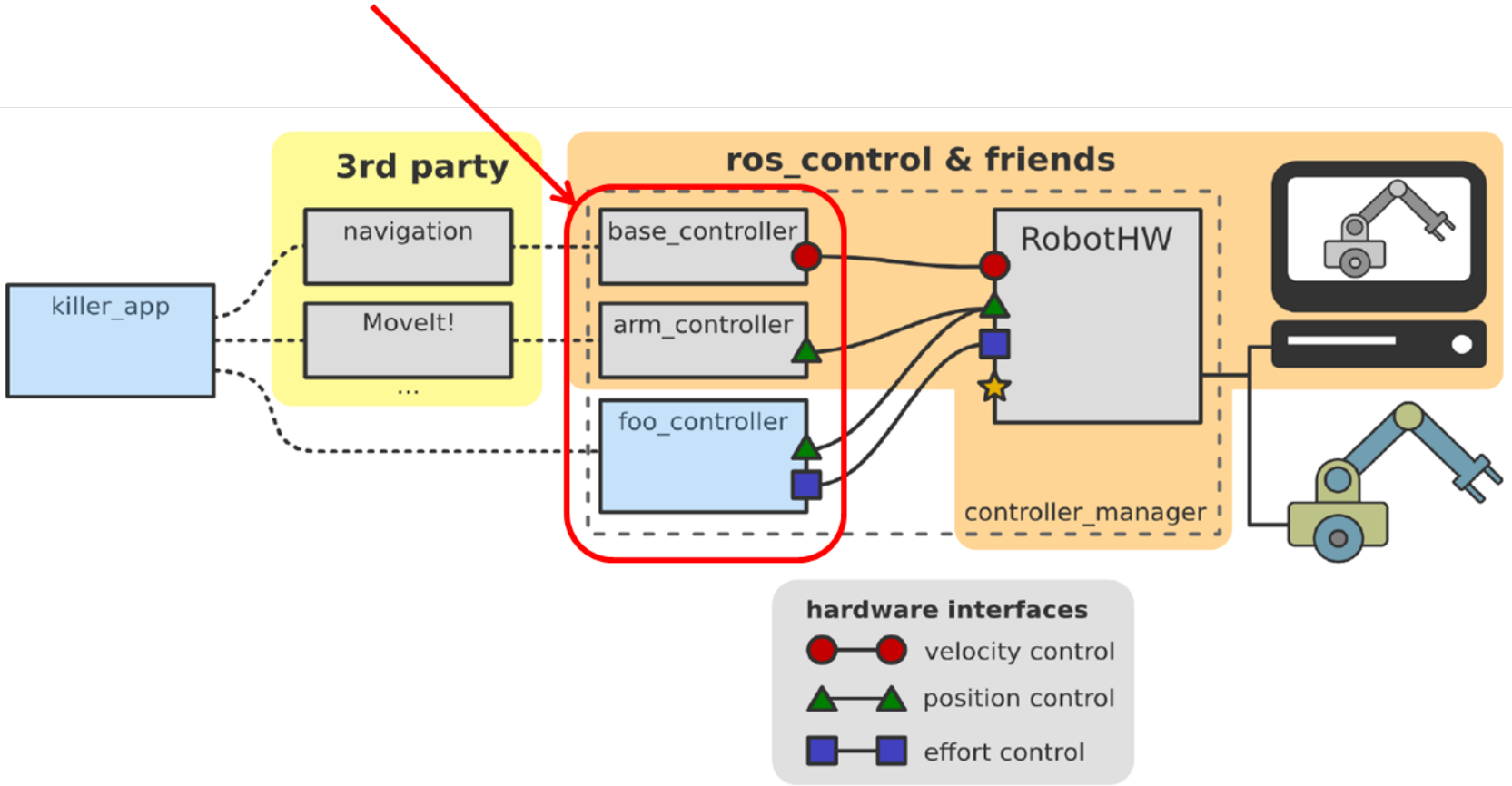
How to customize it for your robot?
How to create reusable code?

What ROS is useful for?



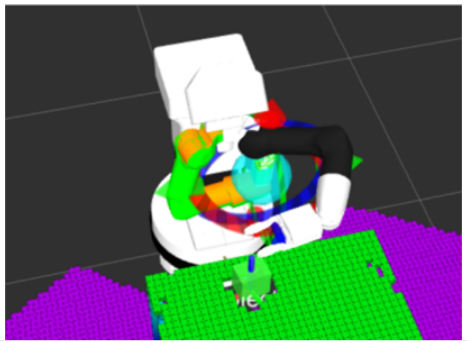
What ROS is useful for?

- **Leverage** standard controllers
- Implement **custom** ones

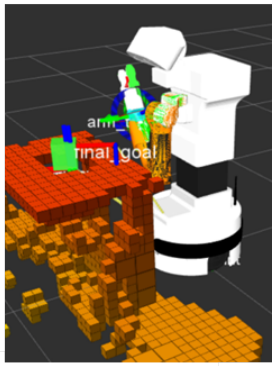


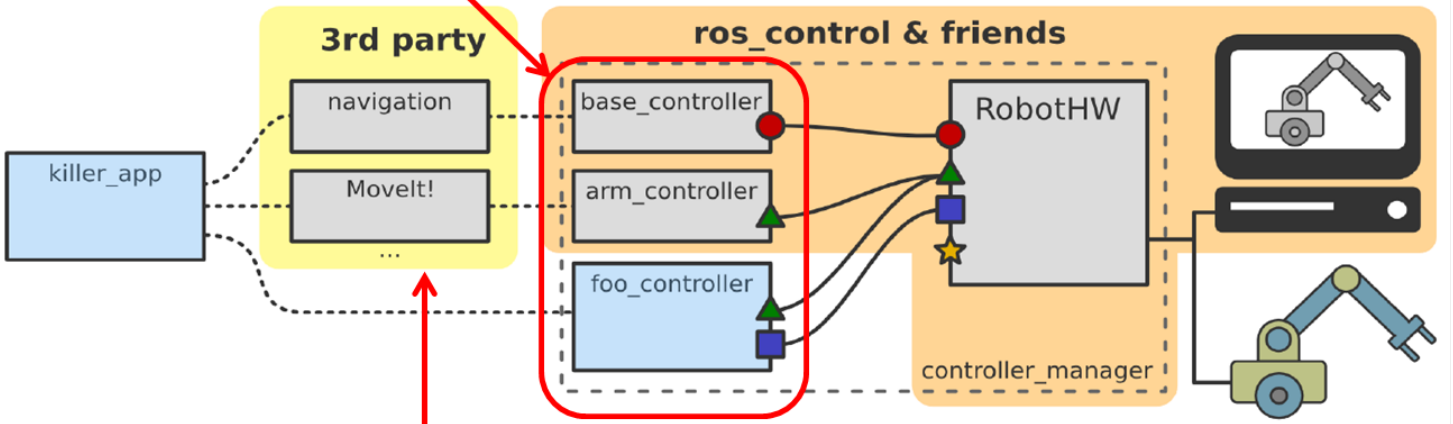
What ROS is useful for?

- **Leverage** standard controllers
- Implement **custom** ones

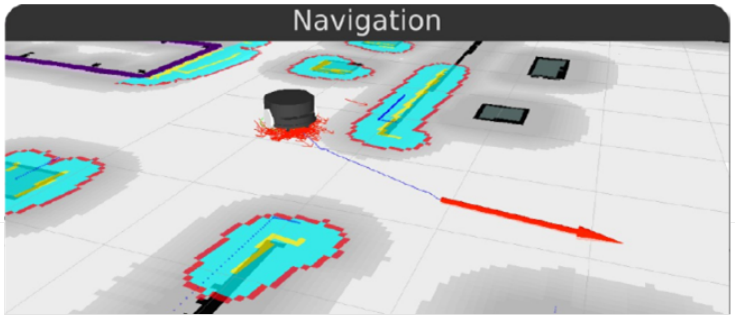


MoveIt!



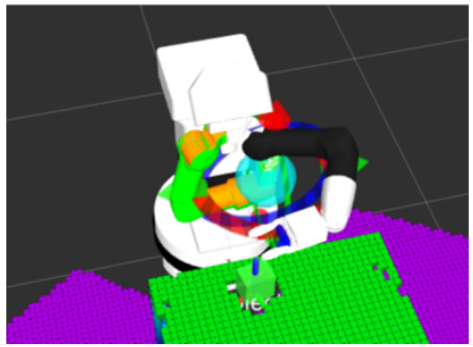


Out-of-the-box
compatibility with
3rd party tools

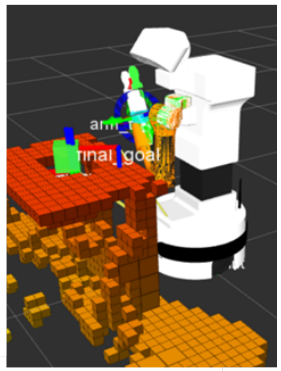


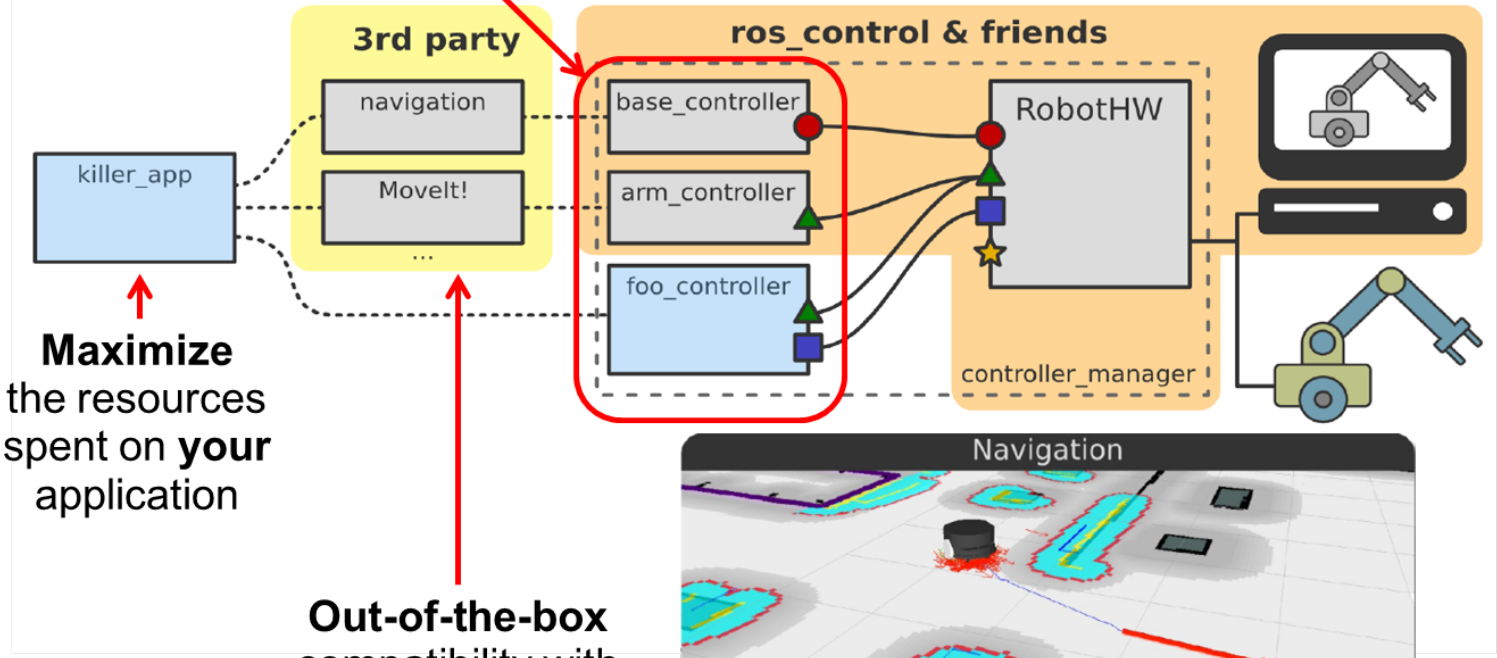
What ROS is useful for?

- **Leverage** standard controllers
- Implement **custom** ones



MoveIt!





Maximize
the resources
spent on **your**
application

Out-of-the-box
compatibility with
3rd party tools

ROS Overview

ROS.org

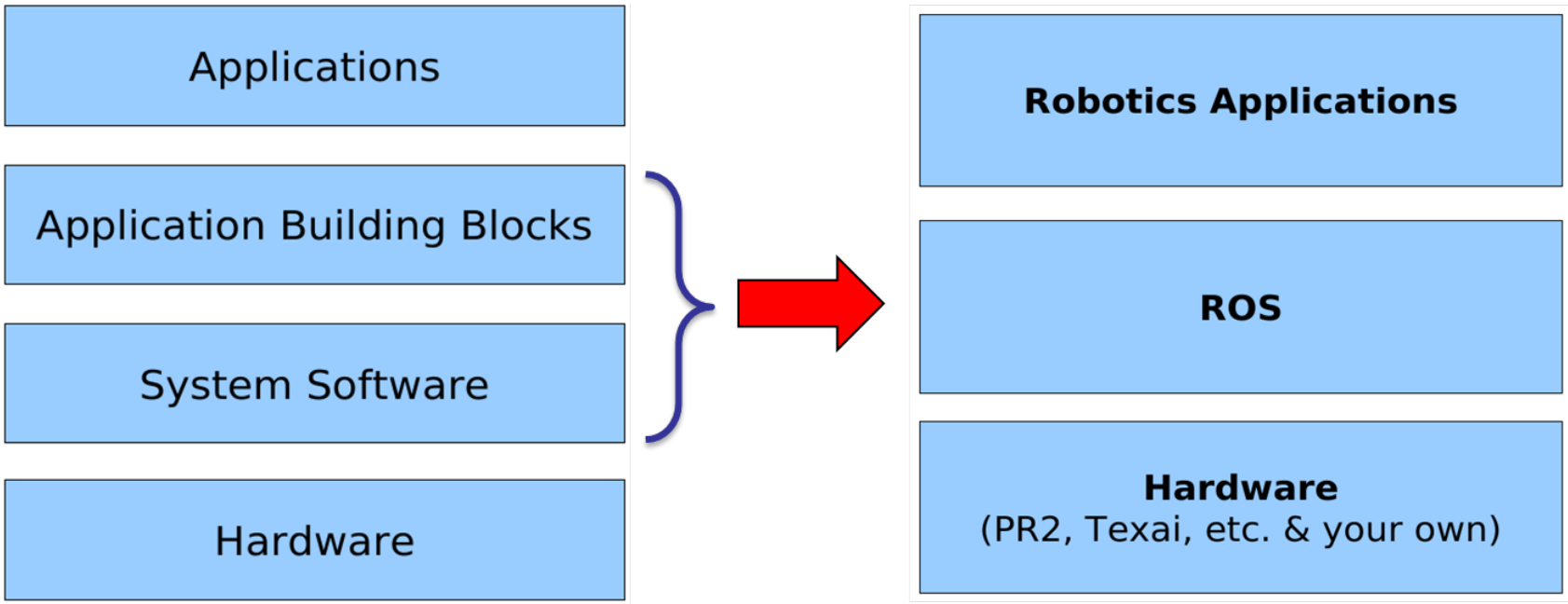
- ROS = Robot Operating System
- ROS is a peer-to-peer robot middleware package
- Supports all major host operating systems

(Ubuntu recommended) Linux | Mac OS X | Windows | Raspbian | QNX

- Ubuntu 22.04 LTS and ROS 2 Humble
- ROS allows for easier hardware abstraction and code reuse
- Functionalities are broken up into a number of chunks that communicate with each other using messages
- Each chunk is called "node" and typically runs as a separate process
- Distributed applications made easy
- ROS users forum: http://answers.ros.org

Comparison with PC Ecosystem

- Standardized Layers
- System software abstracts hardware
- Applications leverage other applications (e.g. database, web server)
- A huge sets of libraries



What is ROS?

Plumbing

- Process management
- Inter-process communication
- Device drivers

Tools

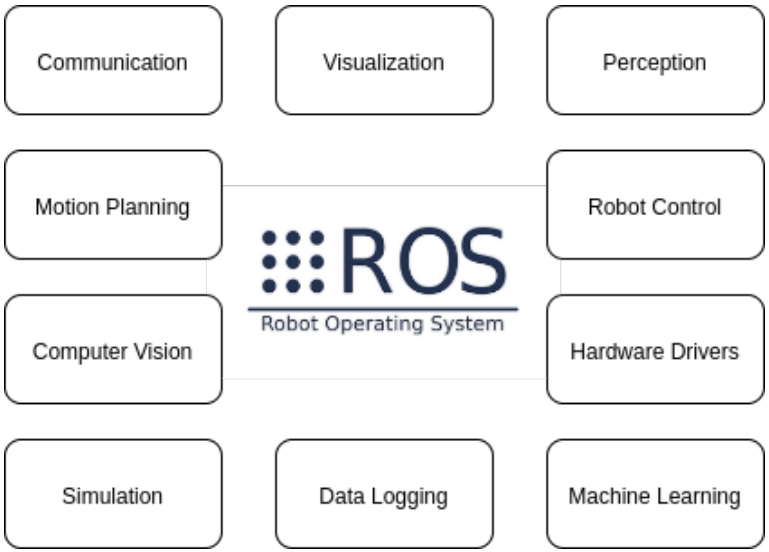
- Graphical user interface
- Simulation
- Distributed parameters
- Visualization
- Data logging

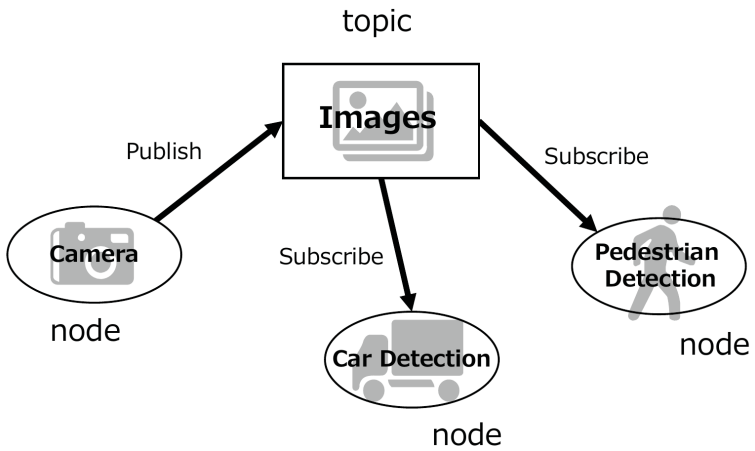
Capabilities

- Control
- Planning
- Perception
- Mapping
- Navigation
- Manipulation

Ecosystem

- Package organization
- Software distribution
- Documentation
- Tutorials
- Testing





ROS for Developers

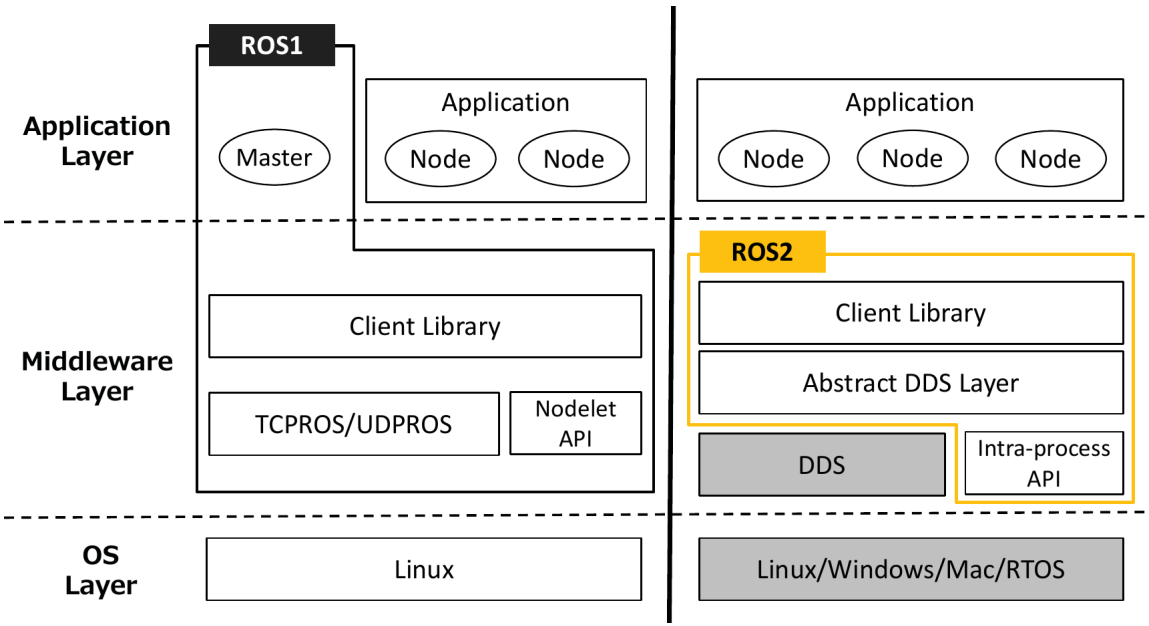
- Native languages: C++ and Python
- Other supported languages. MatLab, Labview, ...
- The code reuse units in ROS are packages
- Package = self-contained directory containing sources, scripts, models, makefiles, builds, etc.
- A large variety of packages can be found
  - sensor drivers, simulators, SLAM, navigation, image processing, etc.
- Integration with other libraries
  - KDL (Kinematic and Dynamic Library for manipulator control)
  - OpenCV (computer vision)
  - PCL (point cloud library)
  - FCL (flexible collision library)
  - OpenAI (Machine Learning)
  - ...

ROS 1 vs. ROS 2

ROS 1	**ROS 2**
Uses TCPROS (custom version of TCP/IP) communication protocol	Uses DDS (Data Distribution System) for communication
Uses ROS Master for centralized discovery and registration. Complete communication pipeline is prone to failure if the master fails	Uses DDS distributed discovery mechanism. ROS 2 provides a custom API to get all the information about nodes and topics
ROS is only functional on Ubuntu OS	ROS 2 is compatible with Ubuntu, Windows 10 and OS X
Uses C++ 11 and Python2	Uses C++ 17 and Python3
ROS only uses CMake build system	ROS 2 supports other build systems
Has a combined build for multiple packages invoked using a single CMakeLists.txt	Supports isolated independent builds for packages to better handle inter-package dependencies
Data Types in message files do not support default values	Data types in message files can now have default values upon initialization
roslaunch files are written in XML with limited capabilities	roslaunch files in Python to support configurable and conditioned execution
Cannot support real-time behavior deterministically even with real-time OS	Supports real-time response with apt RTOS like RTPREEMPT

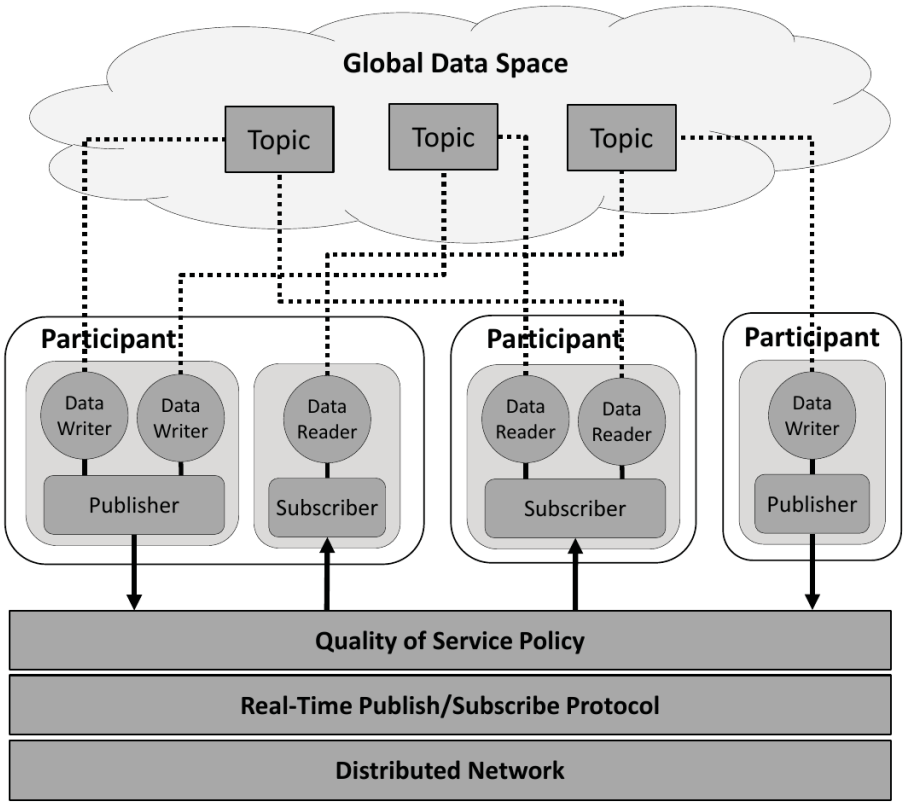
ROS 2 Architecture

- ROS 2 provides language-specific API (rclcpp or rclpy) to support applications
- Below it is C-based ROS 1 client library (rcl), which is used to ensure core algorithms in all language-specific client libraries
- ROS 1 middleware interface (rmw) aims to abstract DDS implementations and streamline Quality-of-Service (QoS) configuration

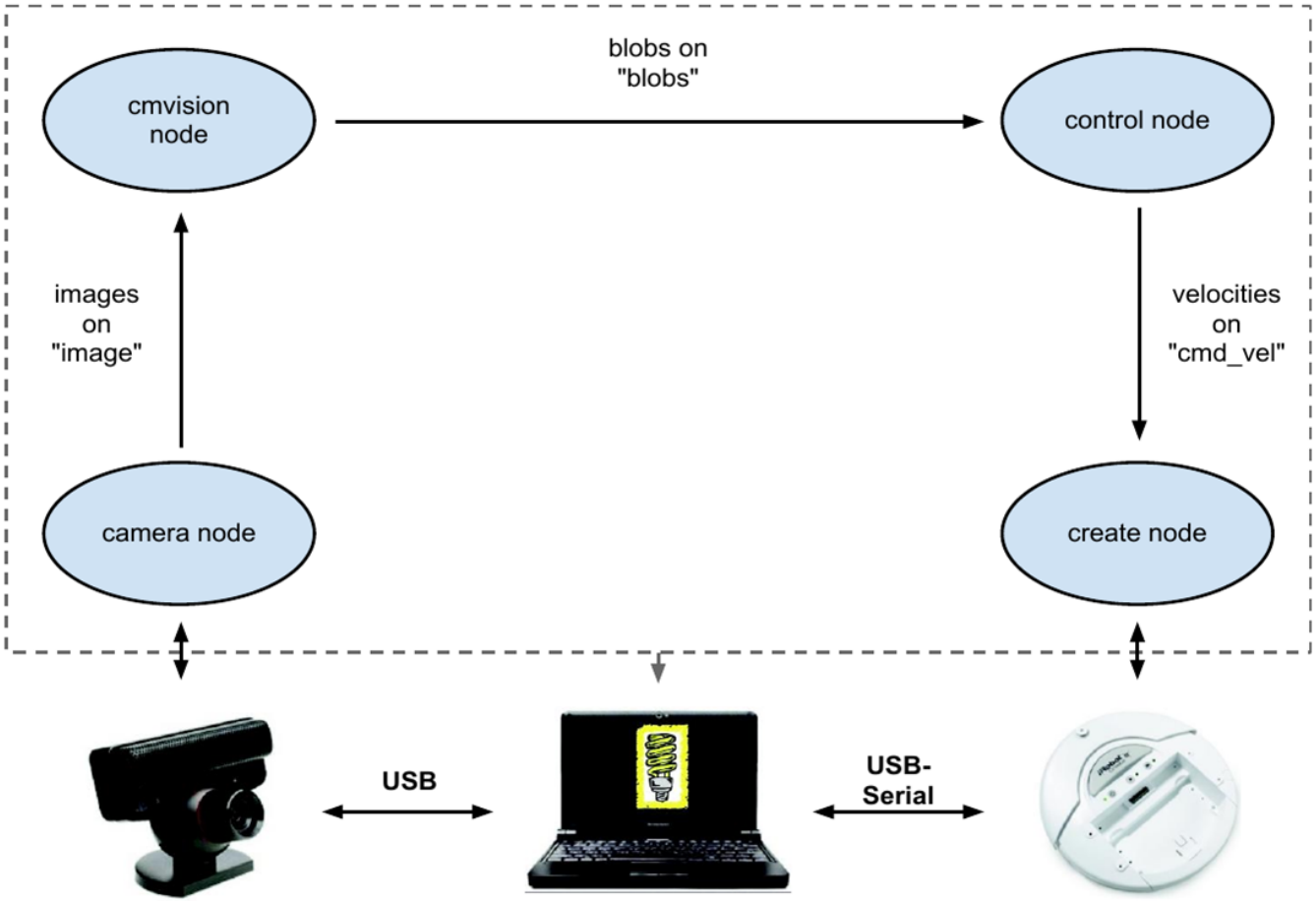


ROS 2 Communication Model

- **Participant**, each publisher or subscriber
- **Publisher**, associated to multiple data writers
- **Subscriber**, associated to multiple data readers
- **DataWriter** to a specific topic (publisher in ROS 1)
- **DataReader** from a specific topic (subscriber in ROS 1)
- **Topic** needs a name and a data type, can store historical message data
- **Quality of Service**: controls all aspects of the communication mechanism (time limit, reliability, continuity, and history)



How ROS works



Basic Example

Install the turtlesim package:

```
$ sudo apt update
$ sudo apt install ros-humble-turtlesim
```

Check that the package installed:

```
$ ros2 pkg executables turtlesim
```

The above command should return a list of turtlesim's executables:

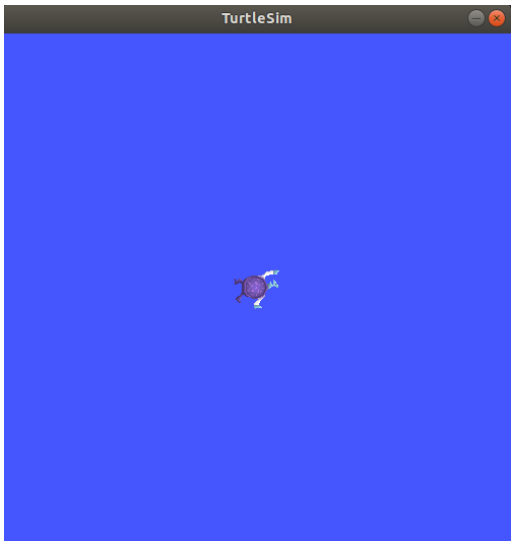
```
turtlesim draw_square
turtlesim mimic
turtlesim turtle_teleop_key
turtlesim turtlesim_node
```

Turtlesim

To start turtlesim, enter the following command in your terminal:

```
$ ros2 run turtlesim turtlesim_node
```

The simulator window should appear, with a random turtle in the center



Here you can see your default turtle's name is `turtle1`, and the default coordinates where it spawns

Open a new terminal and source ROS 2 again

Now you will run a new node to control the turtle in the first node:

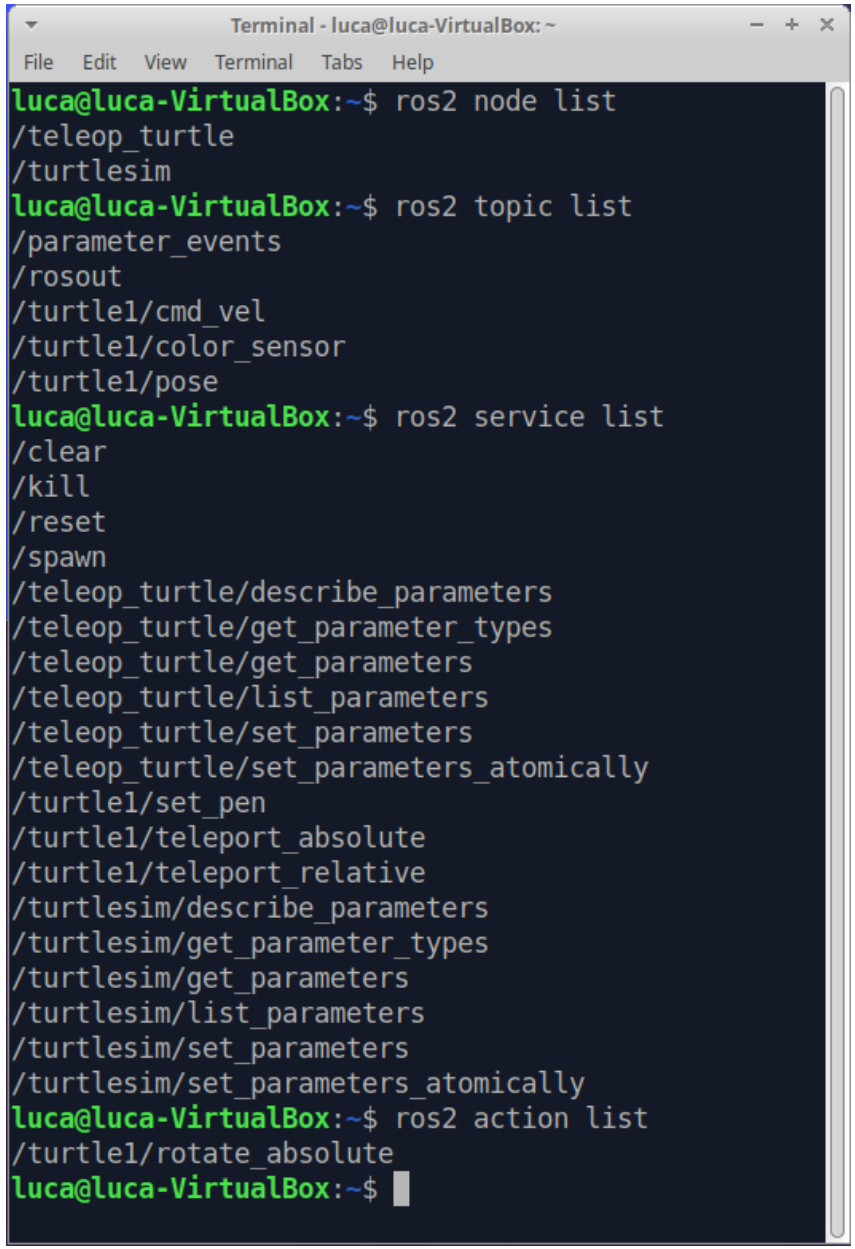
```
$ ros2 run turtlesim turtle_teleop_key
```

Use the arrow keys on your keyboard to control the turtle. It will move around the screen, using its attached "pen" to draw the path it followed so far.

ROS Objects Listing

You can see the nodes and their associated services, topics, and actions using the `list` command:

```
$ ros2 node list
$ ros2 topic list
$ ros2 service list
$ ros2 action list
```

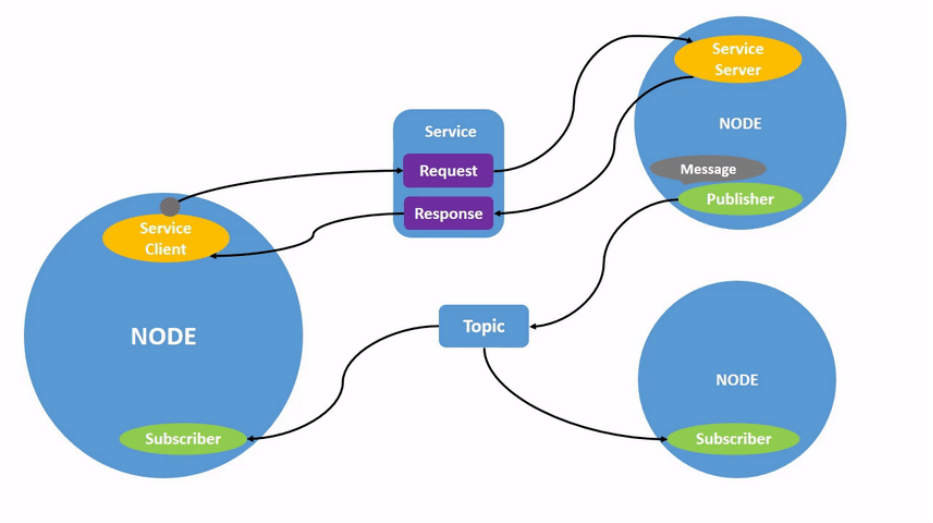


Nodes in ROS 2

A node is a fundamental ROS 2 element that serves a single, modular purpose in a robotics system (e.g. one node for controlling wheel motors, one node for controlling a laser range-finder, etc)

Each node can send and receive data to other nodes via topics, services, actions, or parameters

A full robotic system is comprised of many nodes working in concert. In ROS 2, a single executable (C++ program, Python program, etc.) can contain one or more nodes



Now that you know the names of your nodes, you can access more information about them with:

```
$ ros2 node info <node_name>
```

Let's examine `my_turtle` node info

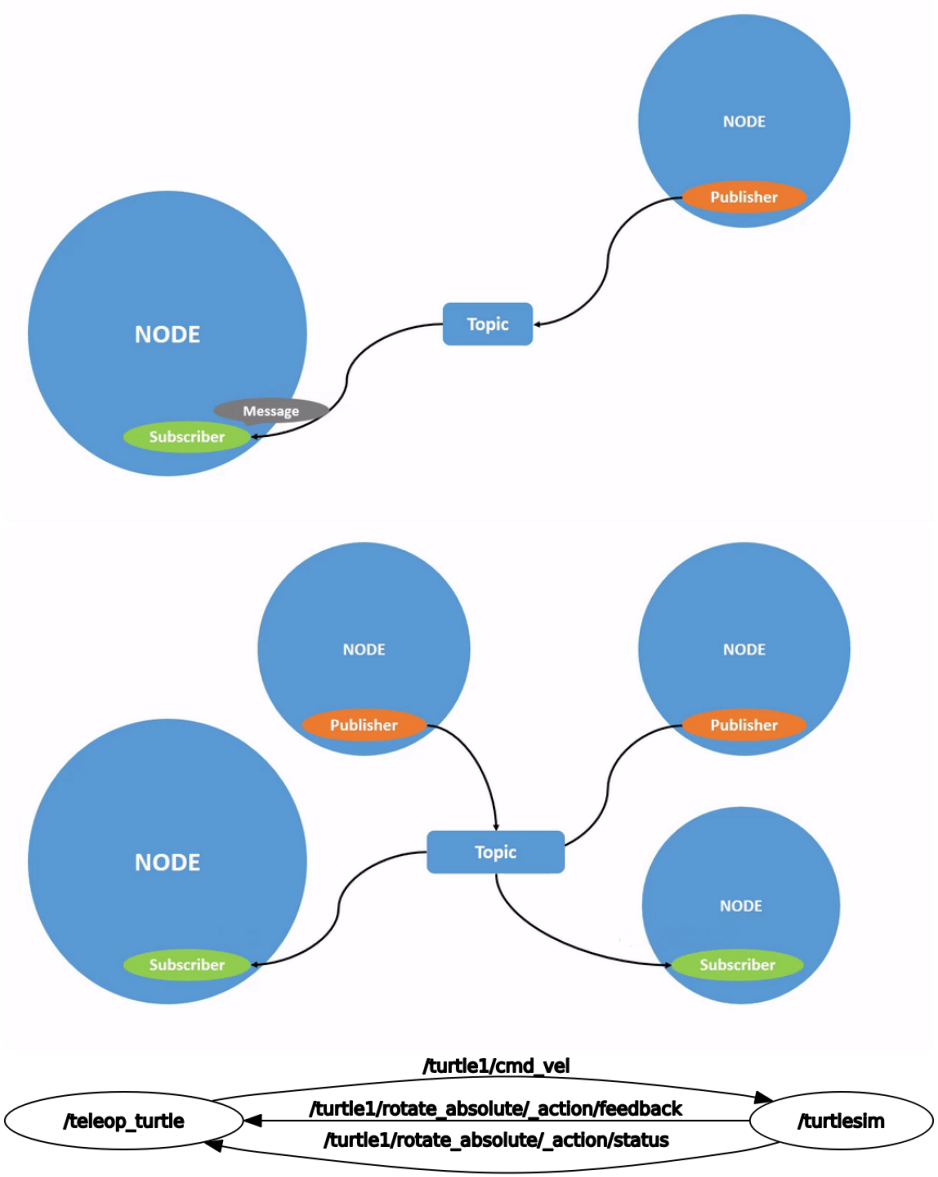
Understanding ROS 2 Topics

Topics are a vital element of the ROS graph that act as a bus for nodes to exchange messages

A node may publish data to any number of topics and simultaneously have subscriptions to any number of topics

Topics are one of the main ways in which data is moved between nodes and therefore between different parts of the system

Use rqt_graph to visualize the changing nodes and topics, as well as the connections between them



Handling ROS 2 Topics

`ros2 topic list -t` will return the list of topics and related type

To see the data being published on a topic, use

```
$ ros2 topic echo <topic_name>
```

Topics don't have to only be point-to-point communication; it can be one-to-many, many-to-one, or many-to-many

```
$ ros2 topic info /turtle1/cmd_vel
```

Which will return

```
Type: geometry_msgs/msg/Twist
Publisher count: 1
Subscription count: 2
```

Now we can run `interface show <msg type>` on this type to learn its details, specifically, what structure of data the message expects

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

Handling ROS 2 Topics (Cont.)

Now that you have the message structure, you can publish data onto a topic directly from the command line using

```
$ ros2 topic pub <topic_name> <msg_type> '<args>'
```

The arguments needs to be input in YAML syntax

```
$ ros2 topic pub --once /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: 1.8}}"
```

Exercise: use the `--rate 1` option, which tells ros2 topic pub to publish the command in a steady stream at 1 Hz

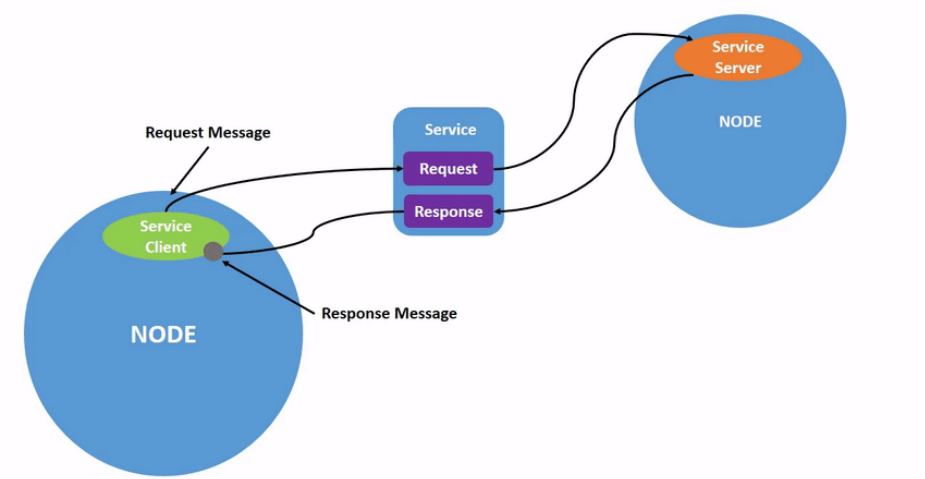
You can view the rate at which data is published using

```
$ ros2 topic hz /turtle1/pose
```

Understanding ROS 2 Services

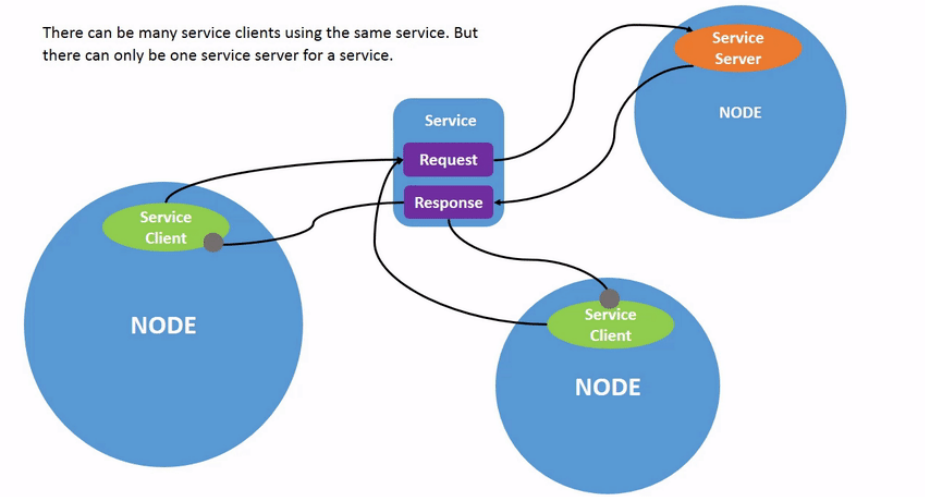
Services are based on a call-and-response model, versus topics' publisher-subscriber model

While topics allow nodes to subscribe to data streams and get continual updates, services only provide data when they are specifically called by a client



To find out the type of a service, use the command:

```
$ ros2 service type <service_name>
```



To see the types of all the active services at the same time, use the `-t` option

```
$ ros2 service list -t
```

You can see the structure of the input arguments by using

```
$ ros2 interface show <type_name>
```

Handling ROS 2 Services

Now that you know what a service type is, how to find a service's type, and how to find the structure of that type's arguments, you can call a service using:

```
$ ros2 service call <service_name> <service_type> <arguments>
```

The `<arguments>` part is optional. For example, you know that `Empty` typed services don't have any arguments

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

Now let's spawn a new turtle by calling `/spawn` and inputting arguments. Input `<arguments>` in a service call from the command-line need to be in YAML syntax.

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

rqt Usage

To run rqt:

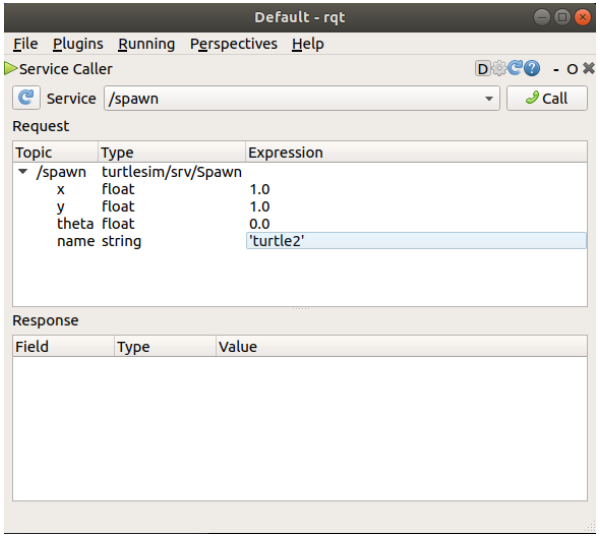
```
$ rqt
```

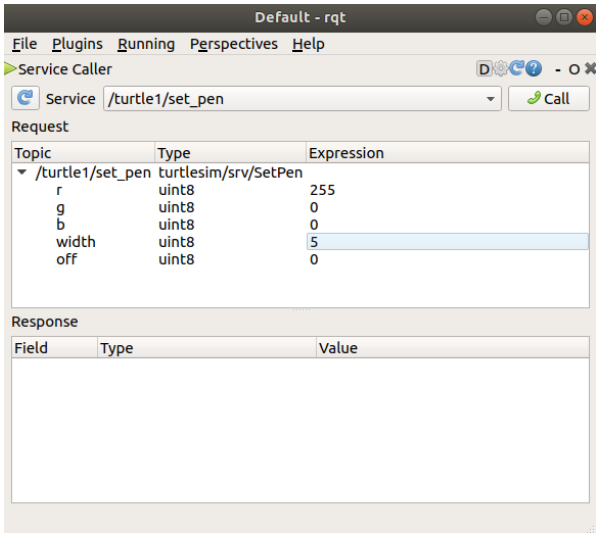
Select **Plugins** > **Services** > **Service Caller** from the menu bar at the top

To spawn turtle2, you have to call the service by clicking the **Call** button

If you refresh the service list in rqt, you will also see that now there are services related to the new turtle, `/turtle2/...`, in addition to `/turtle1/...`

Now let's give turtle1 a unique pen using the `/set_pen` service





Remapping

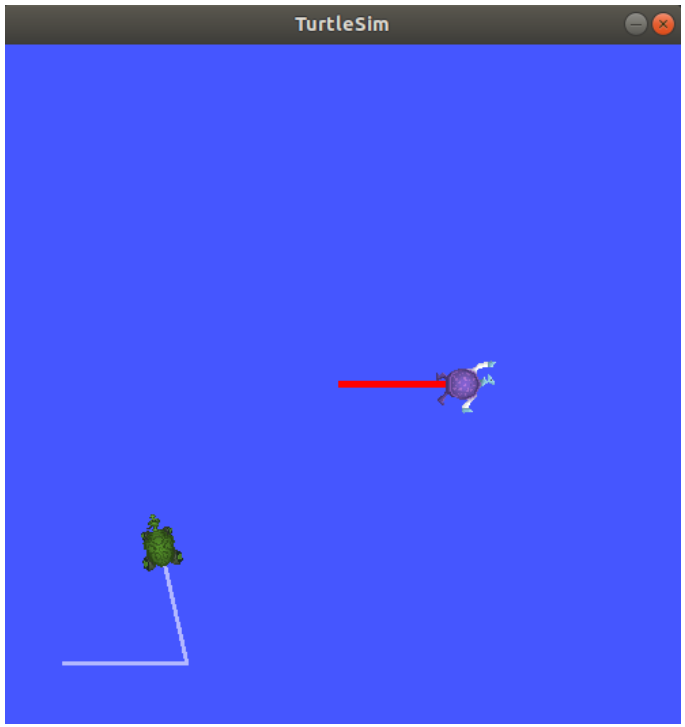
You may wonder if there is a way to move turtle2. You can accomplish this by remapping turtle1's `cmd_vel` topic onto turtle2

In a new terminal, source ROS 2, and run:

```
$ ros2 run turtlesim turtle_teleop_key --ros-args --remap \
    turtle1/cmd_vel:=turtle2/cmd_vel
```

Now you can move turtle2 when this terminal is active, and turtle1 when the other terminal running the `turtle_teleop_key` is active

To stop the simulation, you can enter `Ctrl + C` in the `turtlesim_node` terminal, and `q` in the `teleop` terminal

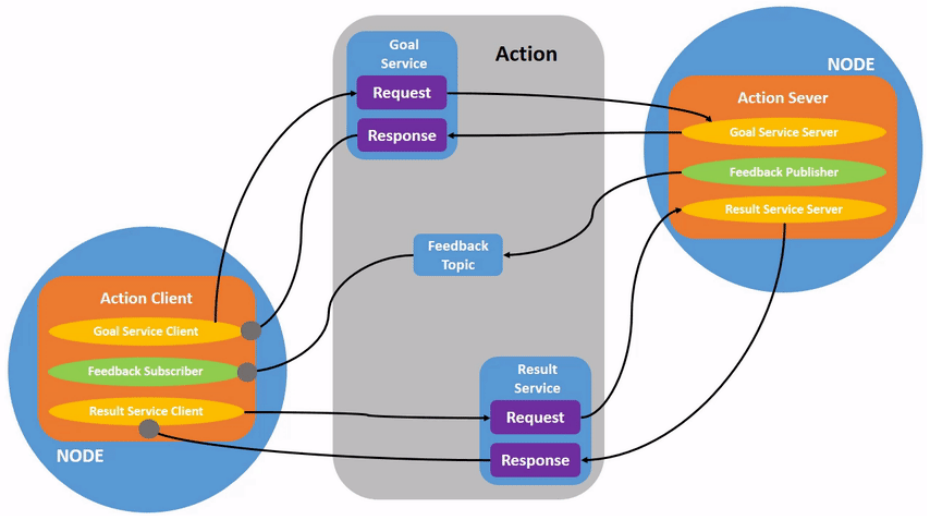


Understanding ROS 2 Actions

Actions are one of the communication types in ROS 2 and are intended for long running tasks

They consist of three parts: a goal, feedback, and a result

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



- Actions use a client-server model, similar to the publisher-subscriber model
- 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

Handling ROS 2 Actions

To identify all the actions and type in the ROS graph, run the command:

```
$ ros2 action list -t
```

Which will return:

```
/turtle1/rotate_absolute [turtlesim/action/RotateAbsolute]
```

You can further introspect the `/turtle1/rotate_absolute` action with the command:

```
$ ros2 action info /turtle1/rotate_absolute
```

Which will return:

```
Action: /turtle1/rotate_absolute
Action clients: 1
    /teleop_turtle
Action servers: 1
    /turtlesim
```

The `/teleop_turtle` node has an action client and the `/turtlesim` node has an action server for the `/turtle1/rotate_absolute` action

Handling ROS 2 Actions (Cont.)

One more piece of information you will need before sending or executing an action goal yourself is the structure of the action type

```
$ ros2 interface show turtlesim/action/RotateAbsolute
```

Which will return:

```
# 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
```

The first section of this message, above the `---`, is the structure (data type and name) of the goal request, the next section is the structure of the result, the last section is the structure of the feedback

Handling ROS 2 Actions (Cont.)

Send an action goal from the command line with the following syntax:

```
$ ros2 action send_goal <action_name> <action_type> <values>
```

Add `--feedback` to the `ros2 action send_goal` command:

```
$ ros2 action send_goal /turtle1/rotate_absolute \
 turtlesim/action/RotateAbsolute "{theta: -1.57}" --feedback
```

Your terminal will return the message:

```
Sending goal:
   theta: -1.57

Goal accepted with ID: e6092c831f994afda92f0086f220da27

Feedback:
  remaining: -3.1268222332000732
...
Result:
  delta: 3.1200008392333984

Goal finished with status: SUCCEEDED
```

Understanding ROS 2 Parameters

A parameter is a configuration value of a node. You can think of parameters as node settings. A node can store parameters as integers, floats, booleans, strings, and lists

To see the parameters belonging to your nodes, open a new terminal and enter the command:

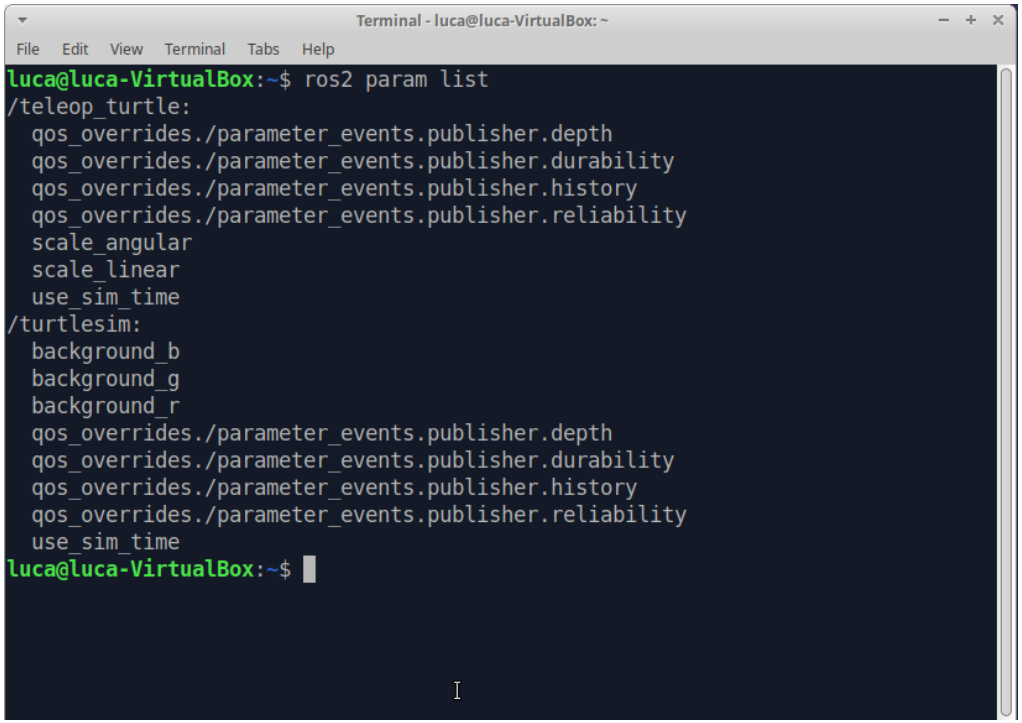
```
$ ros2 param list
```

You will see the node namespaces, `/teleop_turtle` and `/turtlesim`, followed by each node's parameters

Every node has the parameter `use_sim_time`

To determine a parameter's type, you can use

```
$ ros2 param get
```



Handling ROS 2 Parameters

Let's find out the current value of `/turtlesim` parameter `background_g`

```
$ ros2 param get /turtlesim background_g
```

Which will return the value:

```
$ Integer value is: 86
```

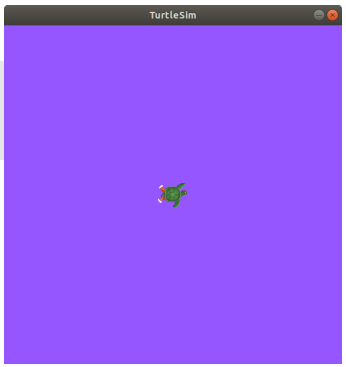
To change a parameter's value at runtime, use the command:

```
$ ros2 param set <node_name> <parameter_name> <value>
```

Let's change `/turtlesim` background color:

```
$ ros2 param set /turtlesim background_r 150
```

The background of your turtlesim window should change colors



Handling ROS 2 Parameters (Cont.)

You can dump all of a node's current parameter values into a file to save them for later by using the command:

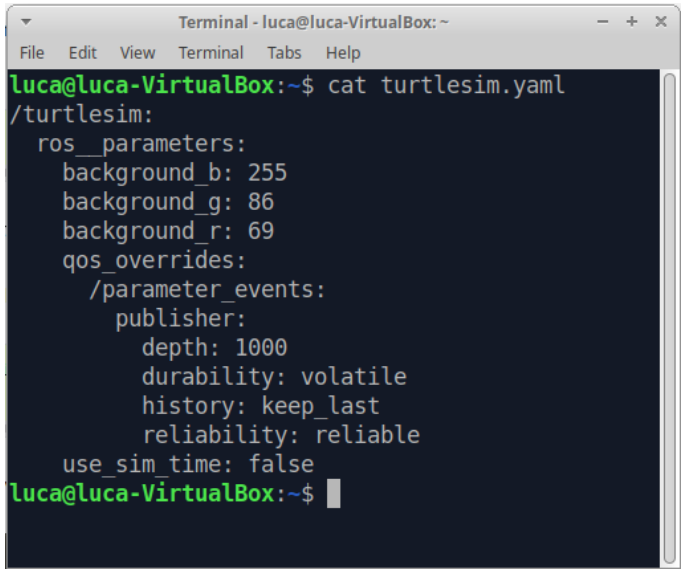
```
$ ros2 param dump /turtlesim > turtlesim.yaml
```

Your terminal will return the message:

```
Saving to: ./turtlesim.yaml
```

You will find a new file in the directory your workspace is running in

Dumping parameters comes in handy if you want to reload the node with the same parameters in the future by using the command




```
$ ros2 param load /turtlesim ./turtlesim.yaml
```

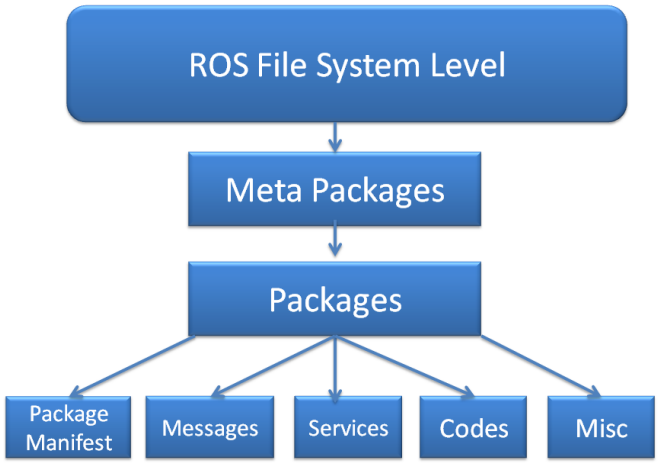
What a Workspace Is?

A workspace is a directory containing ROS 2 packages

You also have the option of sourcing an "overlay" - a secondary workspace where you can add new packages without interfering with the existing ROS 2 workspace that you're extending, or "underlay"

Your underlay must contain the dependencies of all the packages in your overlay

Packages in your overlay will override packages in the underlay. It's also possible to have several layers of underlays and overlays, with each successive overlay using the packages of its parent underlays.



Creating a Workspace

Best practice is to create a new directory for every new workspace

The name doesn't matter, but it is helpful to have it indicate the purpose of the workspace

Let's choose the directory name `dev_ws`, for "development workspace"

```
$ mkdir -p ~/dev_ws/src
$ cd ~/dev_ws/src
```

Another best practice is to put any packages in your workspace into the `src` directory. The above code creates a `src` directory inside `dev_ws` and then navigates into it
Clone a sample repo

```
$ git clone https://github.com/ros/ros_tutorials.git \
    -b humble
```

Now `ros_tutorials` is cloned in your workspace, but it isn't a fully-functional workspace yet. You need to resolve dependencies and build the workspace first

Resolve Dependencies

Before building the workspace, you need to resolve package dependencies

From the root of your workspace directory, run the following command:

```
$ rosdep install -i --from-path src --rosdistro humble -y
```

Packages declare their dependencies in the package.xml file (you will learn more about packages in the next tutorial). This command walks through those declarations and installs the ones that are missing

You can now build your packages using the command:

```
$ colcon build
```

`colcon` is an iteration on the ROS build tools `catkin_make`, `catkin_make_isolated`, `catkin_tools` and `ament_tools`

If `colcon` is not available in your system, install it with

```
$ sudo apt install python3-colcon-common-extensions
```

Creating a Workspace (Cont.)

Once the build is finished, enter `ls` in the workspace root (`~/dev_ws`) and you will see that colcon has created new directories:

```
build install log src
```

The `install` directory is where your workspace's setup files are, which you can use to source your overlay: in a new terminal

```
$ cd ~/dev_ws
$ . install/local_setup.bash
```

Now you can run the turtlesim package from the overlay:

```
$ ros2 run turtlesim turtlesim_node
```

You can modify `turtlesim` in your overlay by editing the title bar on the turtlesim window. To do this, locate the `turtle_frame.cpp` file in `~/dev_ws/src/ros_tutorials/turtlesim/src`.
On line 52 you will see the function `setWindowTitle("TurtleSim");`.
Change the value `"TurtleSim"` to `"MyTurtleSim"`, and save the file.
Return to first terminal where you ran `colcon build` earlier and run it again