Robot Operating System (ROS2)

Vilma Muço

Université de Toulon, Master Ingénierie des Systèmes Complexes (ISC) -Erasmus Mundus MIR

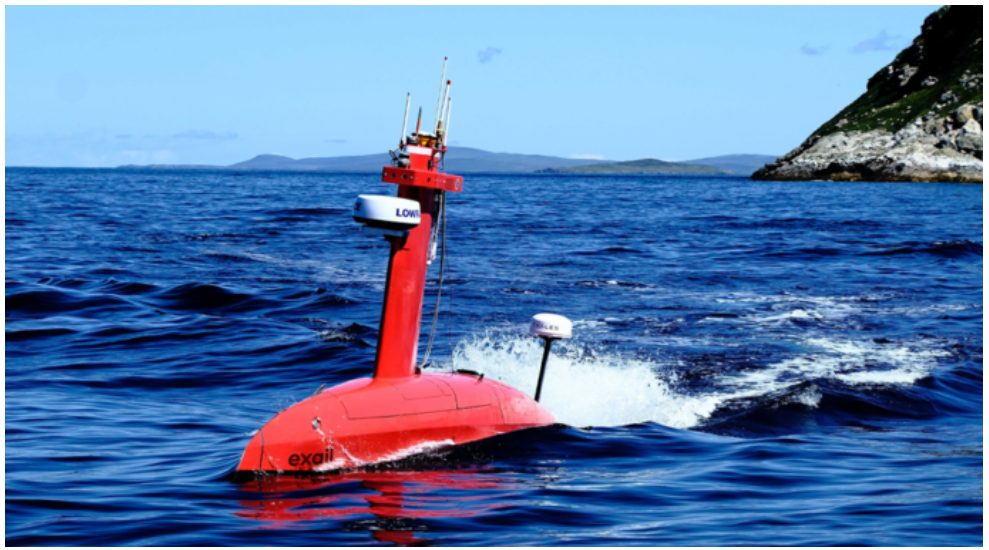
November, 2024

Presentation

- Master of Science in Informatics at Grenoble (MoSIG) - Artificial Intelligence and the Web
- Robotics Engineer in Toulon
- working on DriX

UGA Université Grenoble Alpes

exail



- Master of Science in Informatics at Grenoble (MoSIG) - Artificial Intelligence and the Web
- Robotics Engineer in Toulon
- working on DriX

exail

1. Robotic engineer at Exail
2. working on the Drix USV - the red drone on the screen
3. drivers
4. sensor data
5. aml probe AML - pronounce in english
6. before i did a master

Table of Contents

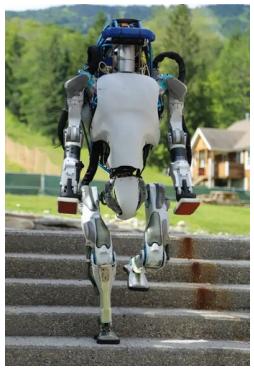
1. In this course you are going to learn ROS 2
2. The goal is to be able to understand and build a robot application
3. use of python as a programming language

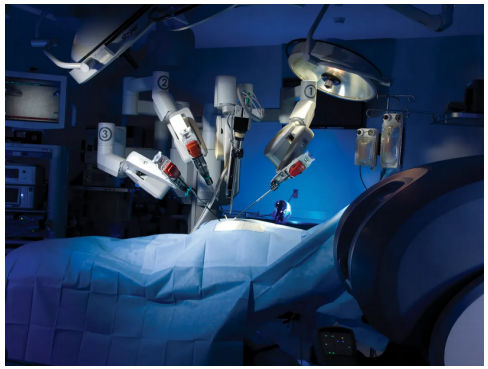
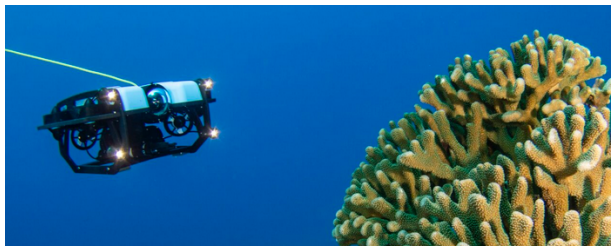
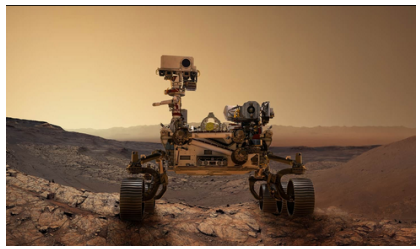
Sources

1. Chapter 1 and 2 of the book Concise Introduction to Robot Programming With Ros2 by Francisco Martín Rico
2. https://docs.ros.org/en/jazzy/Tutorials.html

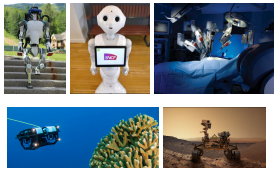
Middleware for robotics

Robot applications $\rightarrow$ Performing tasks in the real world



Robot applications → Performing tasks in the real world



1. during the last decade
2. many robots out
3. build sw for robots is very complex process
4. tasks in real, dynamic and unpredictable world
5. different types and models of sensors
6. example pepper
7. control the head and arms
8. different reference frames
9. middleware provides a set of libraries with state of the art algos
10. not reinvent the wheel

Middlewares for Robotics

- Yarp (humanoid robots)
- Carmen
- Player/Stage
- ROS

Middlewares for Robotics

- Yarp (humanoid robots)
- Carmen
- Player/Stage
- ROS

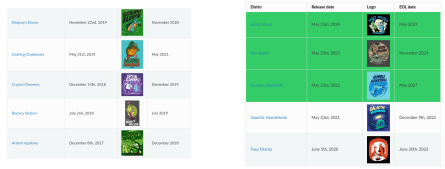
1. During the history of robotics many middlewares have emerged
2. few actually made it ouside the labs and have survived the robot for which they were created
3. yarp - icube humanoid robots, carmen ...
4. stage - sim used with ros (i used during master internship)
5. ros stardand in robotics
6. we have many distributions of ros which are successive versions cut from rolling
7. in this course we are going to study the second generation
8. many concepts are similar with ros1
9. syntax is different and back-ward compatibility is difficult

ROS2 Distributions

Eloquent Elusor	November 22nd, 2019		November 2020
Dashing Diademata	May 31st, 2019		May 2021
Crystal Clemmys	December 14th, 2018		December 2019
Bouncy Bolson	July 2nd, 2018		July 2019
Ardent Apalone	December 8th, 2017		December 2018

Distro	Release date	Logo	EOL date
Jazzy Jalisco	May 23rd, 2024		May 2029
Iron Irwini	May 23rd, 2023		November 2024
Humble Hawksbill	May 23rd, 2022		May 2027
Galactic Geochelone	May 23rd, 2021		December 9th, 2022
Foxy Fitzroy	June 5th, 2020		June 20th, 2023

ROS2 Distributions



1. A distribution is a collection of libraries, applications and tools whose version is verified to work correctly together
2. related to linux distributions
3. as you can see the name increments with every distribution
4. 10 distros of ros 2
5. new t-shirts
6. in green the distros currently maintained
7. turtle as a maskot

Latest Distribution

- Jazzy Jalisco
  - Ubuntu Linux - Noble Numbat (24.04)
  - Windows 10
  - RHEL/Fedora
  - macOS
  - Debian Bookworm

Latest Distribution

- Jazzy Jalisco
  - Ubuntu Linux - Noble Numbat (24.04)
  - Windows 10
  - RHEL/Fedora
  - macOS
  - Debian Bookworm

1. we are going to use the latest distro
2. related to ubuntu 24.04
3. ubuntu 22.04 jammy jellyfish tier 3 support
4. available in windows and macos
5. personal experience with macos

Upcoming Distribution

- There is a new ROS 2 distribution released yearly on May 23rd (World Turtle Day).
- Kilted Kaiju in May 2025

Distro	Release date	Logo	EOL date
Kilted Kaiju	May 2025	TBD	Nov 2026

- Rolling Ridley

Upcoming Distribution

- There is a new ROS 2 distribution released yearly on May 23rd (World Turtle Day).
- Kilted Kaiju in May 2025

Distro	Release date	Logo	EOL date
Kilted Kaiju	May 23rd, 2025	Yes	November, 2026

- Rolling Ridley

1. new release on the world turtle day 23 may
2. Kilted Kaiju
3. rolling ridley special distro
4. serves as a staging area for future stable distributions
5. a collection of the most recent development releases

What is ROS?

The Robot Operating System (ROS) is a set of software libraries (state-of-the-art algorithms), methodologies and tools for building robot applications.

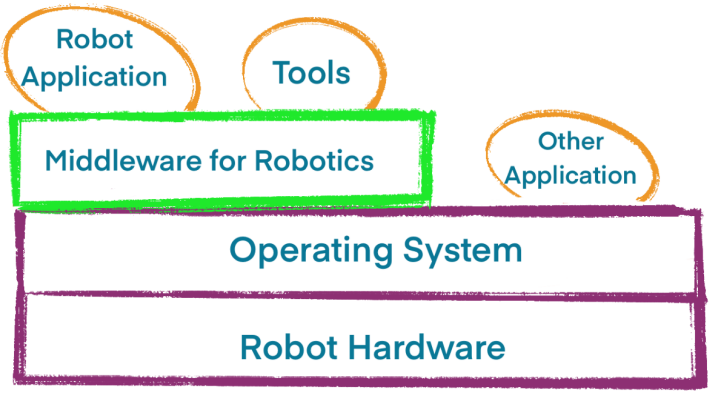


Image adapted from the book *Introduction to Robot Programming With Ros2*

What is ROS?

The Robot Operating System (ROS) is a set of software libraries (state-of-the-art algorithms), methodologies and tools for building robot applications.

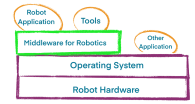


Image adapted from the book *Introduction to Robot Programming With Ros2*

1. acronym
2. difference than os
3. you can see in the schema where the middle ware is situated - green box
4. the orange bubbles represent user applications and that we also have the possibility to use directly the operating system as well
5. the operationg system deals with low level communication with the hardware

Why ROS?

- Free and Open Source
- **Huge community of highly qualified roboticists**

- Free and Open Source
- **Huge community of highly qualified roboticists**

1. As we previously said before that during the history of robotics many middlewares emerged but only ros became so popular as to be considered as a standart
2. why ros became a standart in robotics is mostly due to huge and highly qualified community of people involved
3. many laboratories and notable companies in the field provide applications and libraries and maintaine them with distros

Why ROS?

- Modular Architecture
- Allows to quickly build and easily connect the main robotic components:
  - Actuators
  - Sensors
  - Control Systems
- microRos for real time OS

Why ROS?

- Modular Architecture
- Allows to quickly build and easily connect the main robotic components:
  - Actuators
  - Sensors
  - Control Systems
- microRos for real time OS

1. another way to define robotics is as the art of integration
2. and ros provides us with a lot of tools for integration
3. due to its modular architecture
4. quicky build and connect different actuators sensors etc
5. specific libraries for real time systems
6. microROS - a variant of ROS that runs nativaly on embedded microcontrollers running real time operating systems

Why ROS?

- Tools for monitoring
  - Logs for testing
  - Training
  - Quality Assurance
- Simulated environment and Visualisation Tools

1. another strong point of ros is the monitoring tools
2. logging system and replaying the data for debugging
3. simulation for testing

Planes of study

- The Community
- The workspace (static/ development time)
- The computational Graph (dynamic/ runtime)

- The Community
- The workspace (static/ development time)
- The computational Graph (dynamic/ runtime)

1. approach the study from 3 different dimentions
2. the community is the fondamental element of ros during development
3. provides docs
4. applications and utilities
5. the workspace is the place where the software is installed and where the user programs are situated on the system
6. The computational graph is the view of the running ros2 application
7. different processes might start and stop during runtime - dynamic

ROS Community

- Public repositories
- Development methodology
- Software delivery mechanisms
- Open Robotics
  - https://www.ros.org
  - https://docs.ros.org/en/jazzy/index.html
  - https://robotics.stackexchange.com/
- ROSCon
  - Annual ROS developer conference
  - Regional ROS events like ROSConJP and ROSConFr

ROS Community

- Public repositories
- Development methodology
- Software delivery mechanisms
- Open Robotics
  - https://www.ros.org
  - https://docs.ros.org/en/jazzy/index.html
  - https://robotics.stackexchange.com/
- ROSCon
  - Annual ROS developer conference
  - Regional ROS events like ROSConJP and ROSConFr

1. Open robotics foundation
2. which provides docs, tools applications
3. pages for tutorials, installation guides and how to guides
4. forums for discussions, keeping up to date with the ros community and discussing design issues
5. a page to ask questions
6. license apache 2, BSD
7. there is an annual Ros developer conference
8. as well as regional conferences like ros conf japan or france

Workspaces

Combining workspaces using the shell environment

- Advantages:
  - different distros
  - different vesions of packages

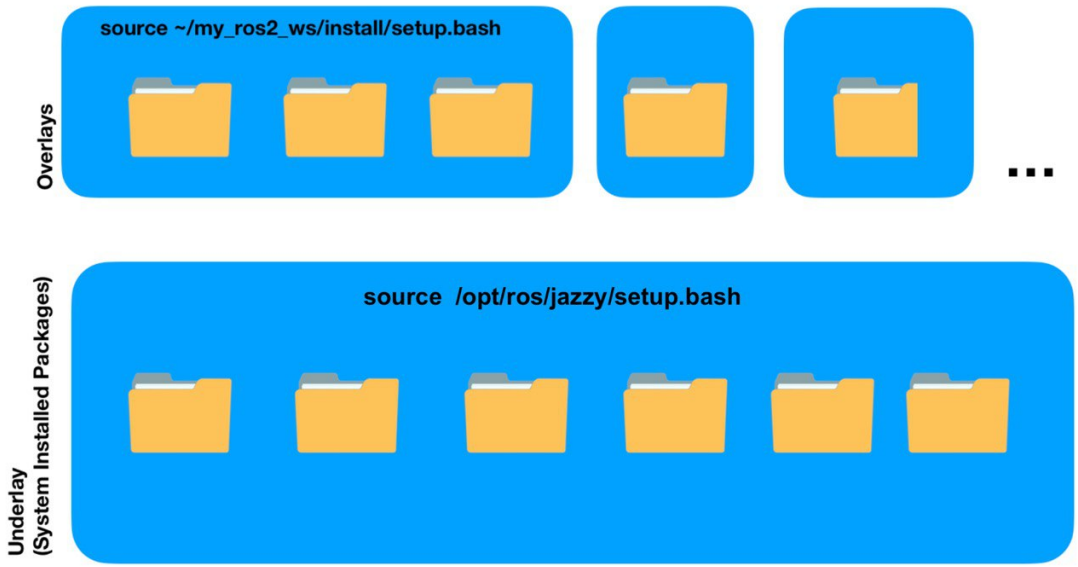
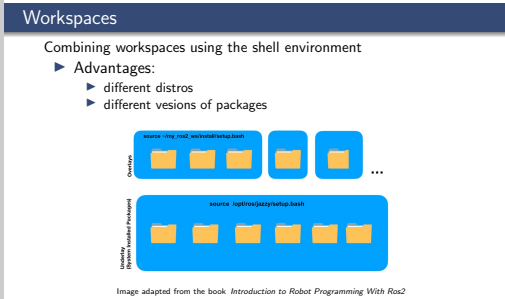
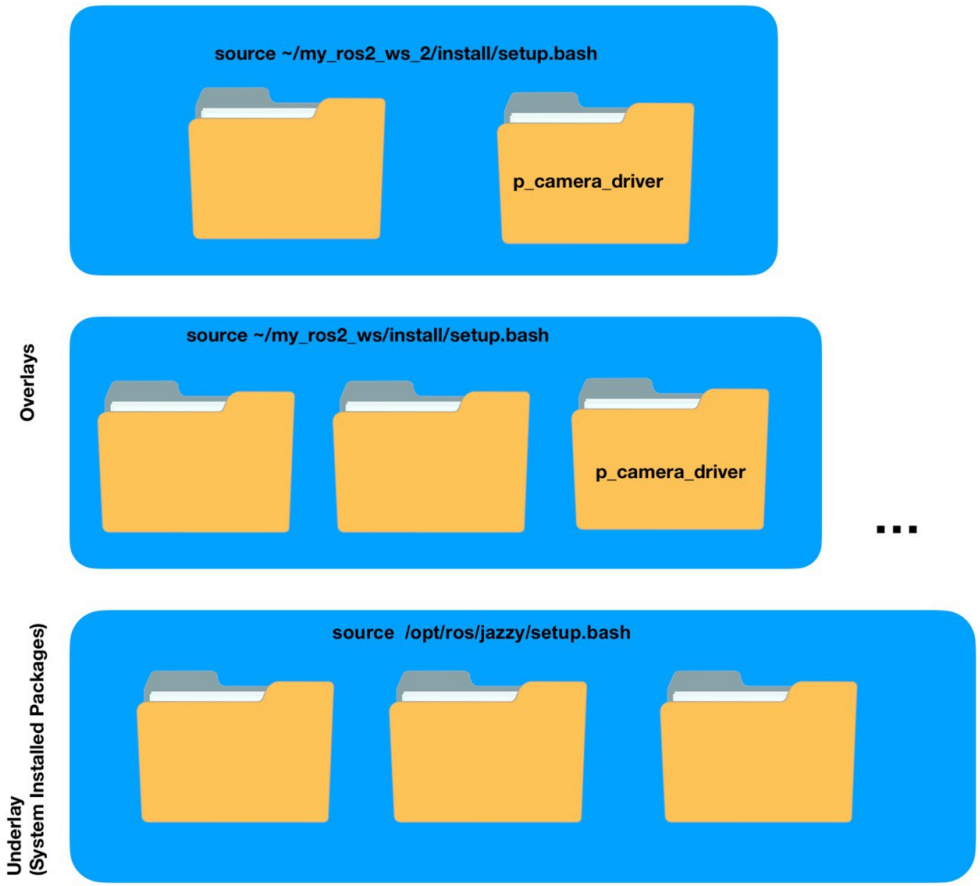


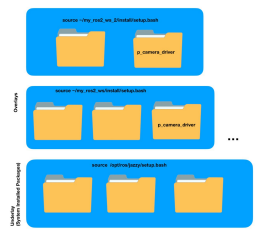
Image adapted from the book *Introduction to Robot Programming With Ros2*



1. the main roscore installation files are also considered as a workspace
2. the user workspaces are called the overlays
3. how we develop in ros is by combining different workspaces through the shell environment
4. we can activate a workspace by sourcing a setup file on the terminal
5. the advantage is that we can install and use different distributions in the same machine
6. another advantage is that we can use different versions of packages by activating different overlays with the same packages

Example Workspaces





1. lets look at this example
2. the package p_camera_driver
3. we can activate the three workspaces
4. only the last package with the same name is active
5. the uderlayer packages are hidden
6. is is quite interesting for testing and develloping different features in parallel

ROS Graph

- The computational Graph (dynamic/ runtime)
  - Nodes
  - Links:
    - Topics
    - Services
    - Actions

ROS Graph

- The computational Graph (dynamic/ runtime)
  - Nodes
  - Links:
    - Topics
    - Services
    - Actions

1. the final dimention is the computational graph
2. if we look a running ros application we can imagine it as a graph with nodes being the elementary processing unit and links,edges, arcs being the ways that nodes communicate
3. the are three main ways that the nodes can use to communicate
4. topics, services and actions

Nodes

- Fundamental ROS element that is responsible for a single, modular purpose in a robotic system.
- Each node sends and receives data from other nodes
- A single executable (ex. python file) can contain one or more nodes

Nodes

- Fundamental ROS element that is responsible for a single, modular purpose in a robotic system.
- Each node sends and receives data from other nodes
- A single executable (ex. python file) can contain one or more nodes

1. Lets look in more details each element of the computational graph
2. The node which is responsible for a single, modular purpose in a robotic system
3. each node can receive and send data from other nodes
4. it is not given that a node is a seperate file a single executable - python script can contain ore or more nodes

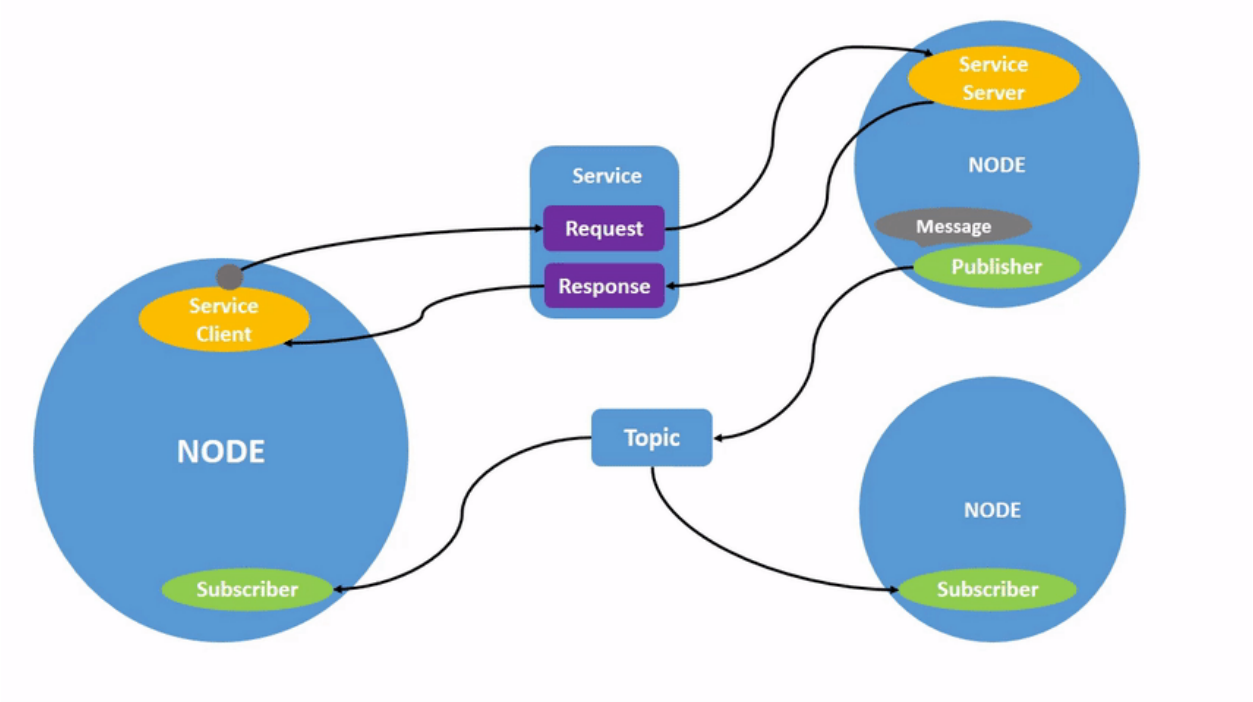


Image taken from *docs.ros.org*

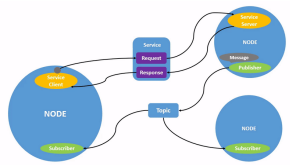


Image taken from *docs.ros.org*

1. here we have a graphical representation of nodes using different types of communications
2. nodes are an elementary element but that doesnt mean that they are not complicated a node can have many links of communication
3. lets look more in details the communications paradigms

- Asynchronous way of transporting messages
- Subscriber and Publisher

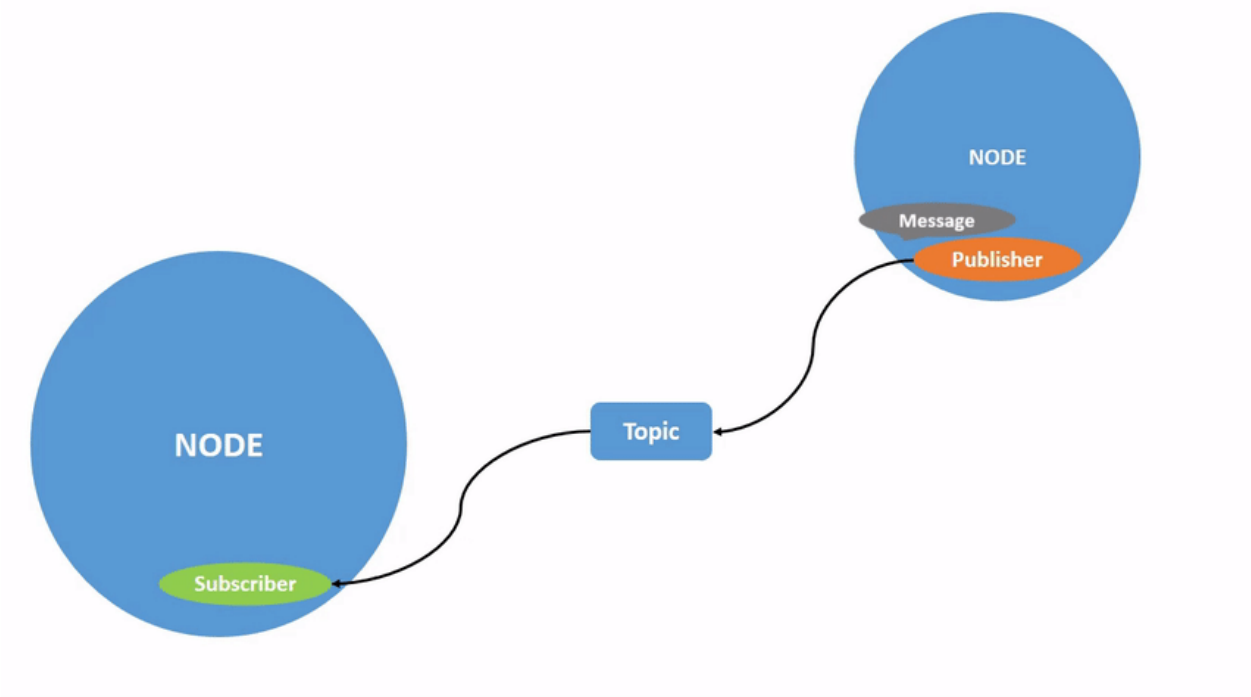


Image taken from *docs.ros.org*

- Asynchronous way of transporting messages
- Subscriber and Publisher

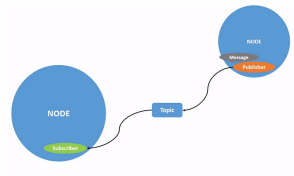


Image taken from *docs.ros.org*

1. Topics are the most common way of communication
2. asynchronous way of transporting messages
3. model of publisher subscriber
4. so the topic serves as a kind of channel which has a specific name
5. and can handle only one kind of message

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

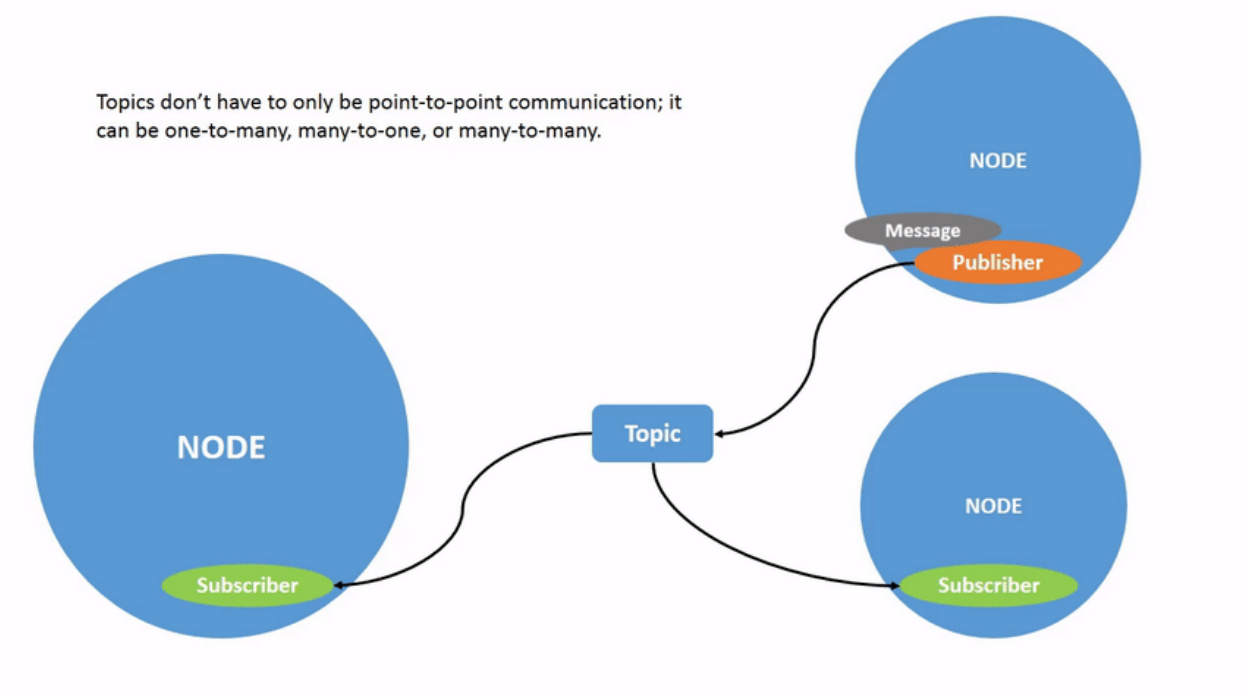


Image taken from *docs.ros.org*

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

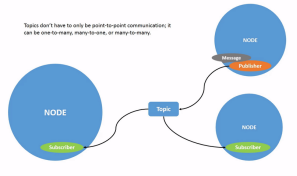
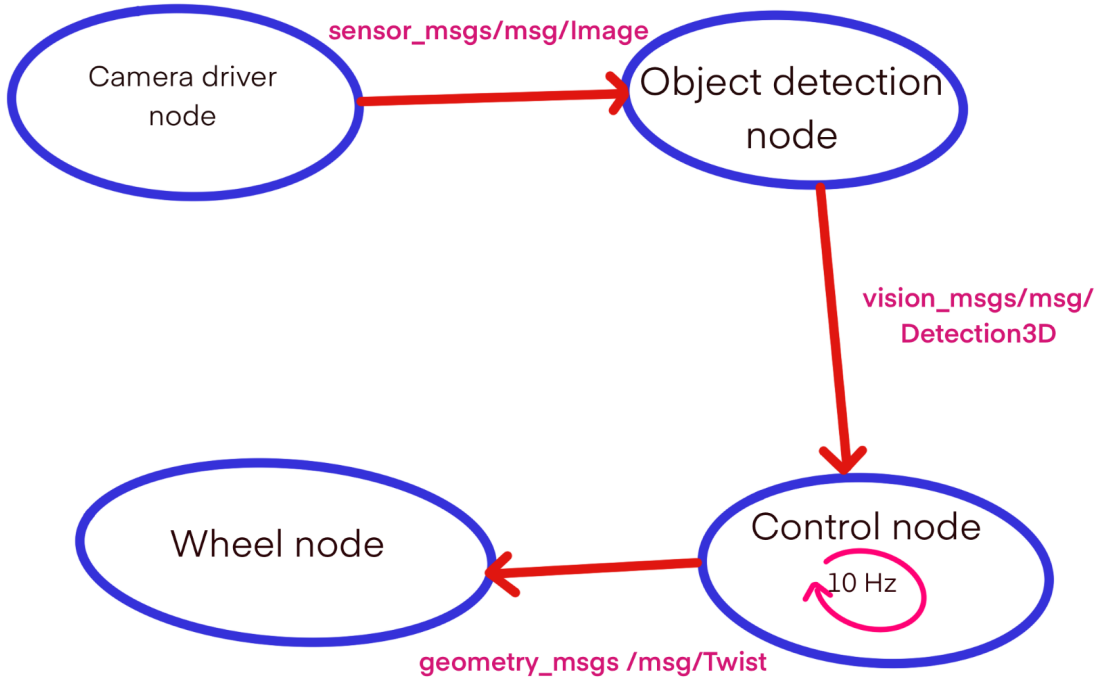


Image taken from *docs.ros.org*

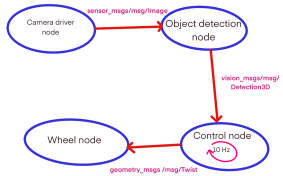
1. we can have many publishers for the same topic as well
2. as well as one or many subscribers that receive the message
3. but you really need to be careful to not create loops

In the example notice:

- Node execution modes
  - Iterative execution
  - Event-oriented execution
- Message types



In the example notice:
- Node execution modes
  - Iterative execution
  - Event-oriented execution
- Message types



1. obstacle avoidance system
2. We have four nodes
3. a camera driver that reads messages udp/tcp and publishes a sensor_msgs/msg/Image to a topic which the object detection is subscribed to
4. The object detection node only publishes when it detects an object
5. The control node is subscribed to the topic the object detection node is publishing and can adjust the velocity commands for the wheel node
6. two modes of execution
7. processing nodes which process information only when they receive it also known as Event-oriented execution
8. example the object detection node

Messages in ROS

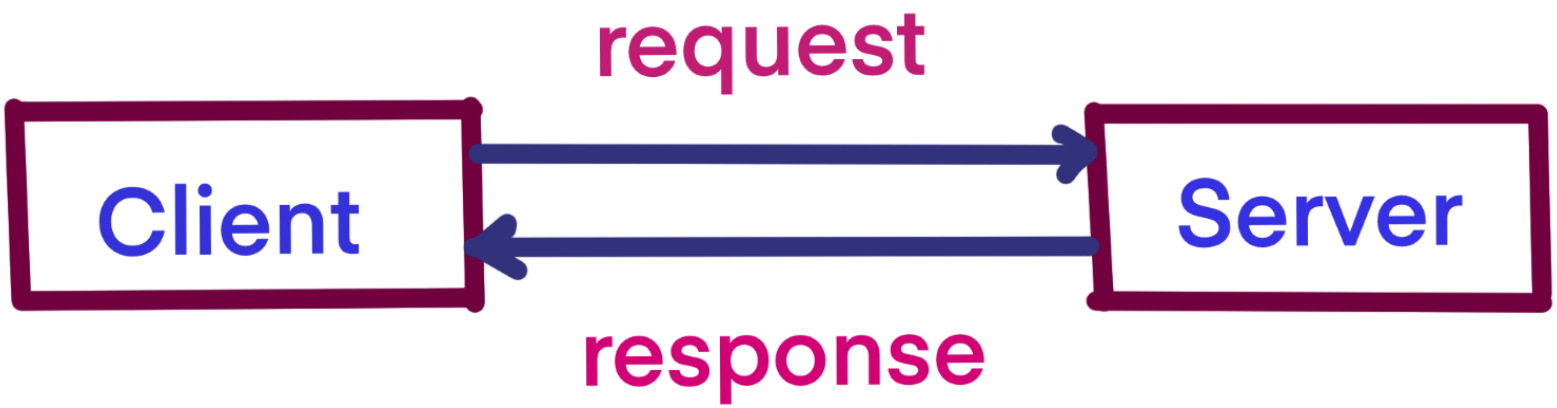
- The names of the resources in ROS follow a convention very similar to the file system in Unix (ex. /turtle1/cmd_vel)
- namespaces to isolate resources
- predefined message types (geometry_msgs/msg/Twist - virtually all robots in ROS2 receive these types of messages to control their speed)
- We can define our own message types in ROS

Messages in ROS

- The names of the resources in ROS follow a convention very similar to the file system in Unix (ex. /turtle1/cmd_vel)
- namespaces to isolate resources
- predefined message types (geometry_msgs/msg/Twist - virtually all robots in ROS2 receive these types of messages to control their speed)
- We can define our own message types in ROS

1. The names of the resources in ros as we saw for the message types follow a convention very similar to the file System in Unix
2. we use namespaces to isolate resources and we can reuse names
3. Ros already provides many standart messages for example the gemetry Twist message to control the velocity of a robot
4. but we can define our own types of messages as wel, which can be quite practical
5. especially combining different standart ones like time message to be used as a timestamp and string for gps frames to know the coordinates at a specific time

- Call-and-response model
- Services only provide data when they are specifically called by a client (Synchronous way of communication).



- Call-and-response model
- Services only provide data when they are specifically called by a client (Synchronous way of communication).

1. The next model of communication is the service
2. based on the client server mode or call-response model
3. these are synchronous mode of communication
4. while a subscriber gets info continuously the client here only gets data when it asks for it
5. this type of communication is used when we need an immediate response
6. because the client asks for data then stops and waits for a response

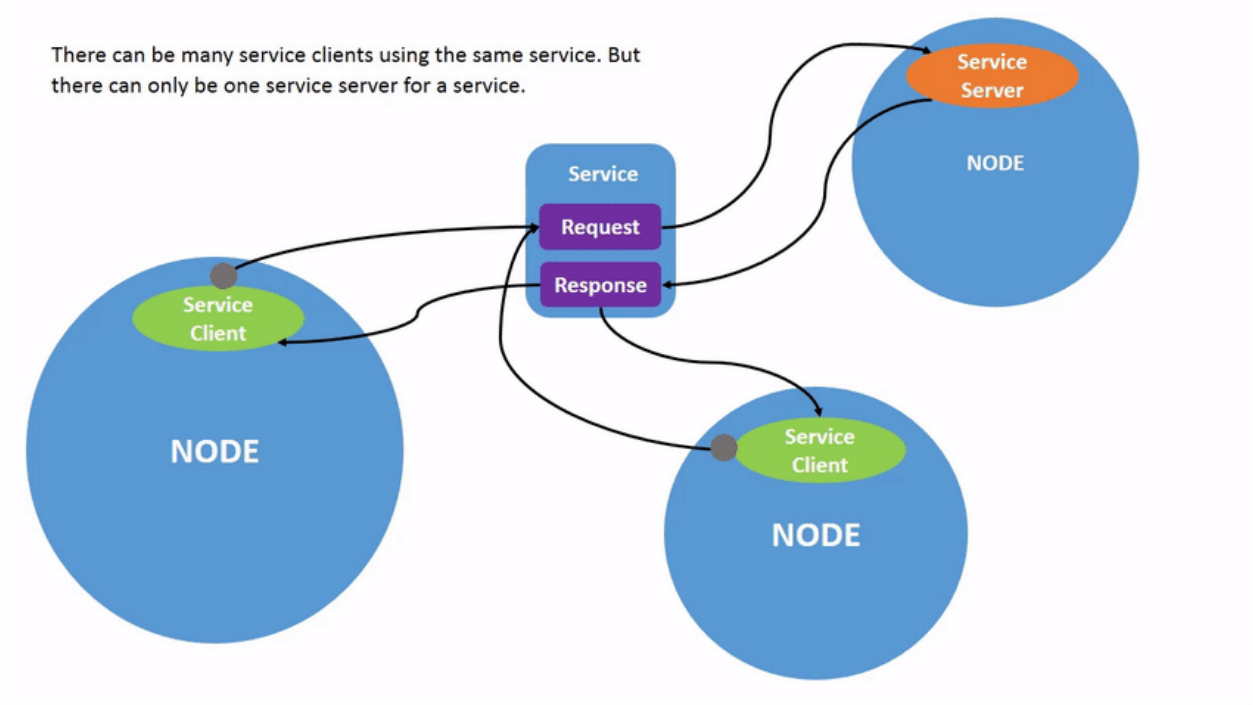


Image taken from *docs.ros.org*

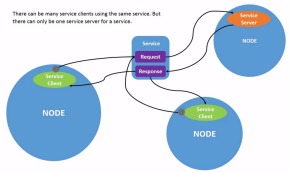
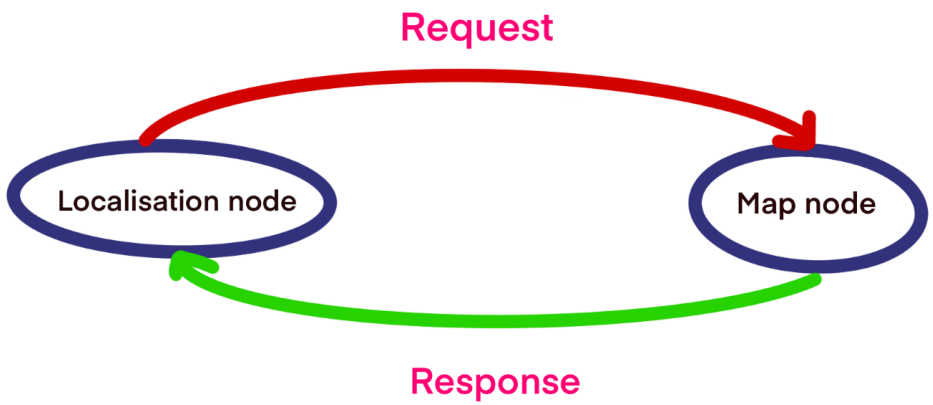


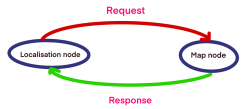
Image taken from *docs.ros.org*

1. we can have multiple client nodes but only one service server or service provider node
2. the request and response message types are predefined

- The call to reset a map with a response if the call succeeded



- The call to reset a map with a response if the call succeeded



1. A concret example of a service is the call to reset the map
2. We can need this for example when we switch off and on a robot or when we move the robot around

- Actions are like services that allow you to execute long running tasks (like navigation), provide regular feedback, and can be canceled.
- They consist of three parts:
  - A goal
  - Feedback
  - 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.

- Actions are like services that allow you to execute long running tasks (like navigation), provide regular feedback, and can be canceled.
- They consist of three parts:
  - A goal
  - Feedback
  - 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.

1. the last way of communation are Actions
2. they are build based on topics and services
3. they are used for long tasks because they allow you to make a request and not stop and wait for the reply but carry on and regular feedback until the task is finished or cancelled
4. they consist of 3 parts
5. a goal
6. feedback
7. a result

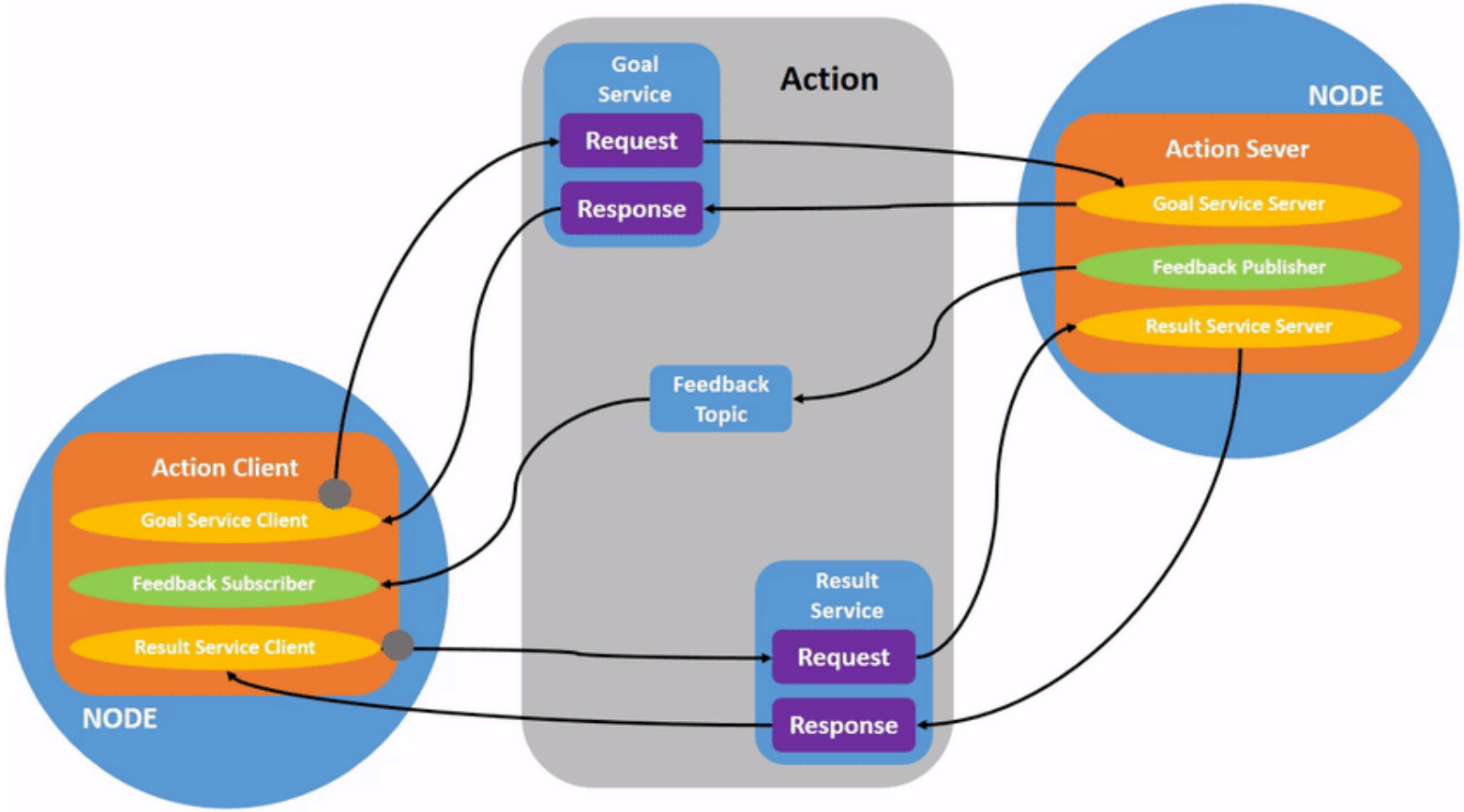


Image taken from *docs.ros.org*

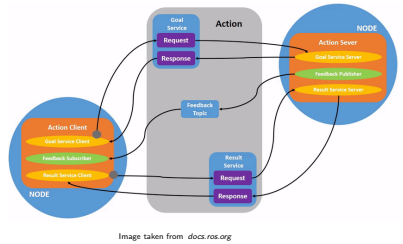
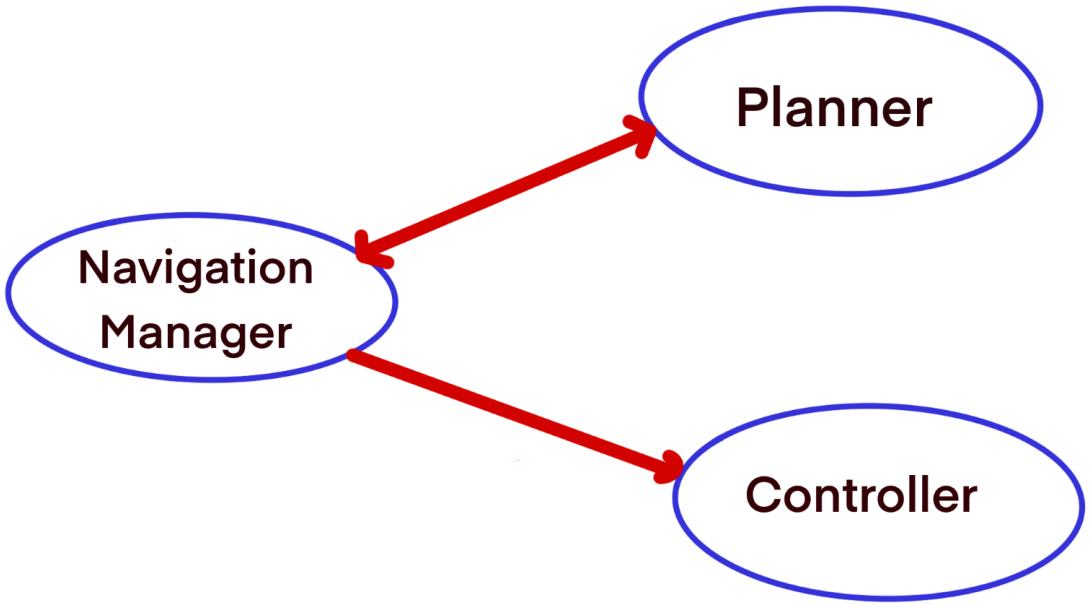


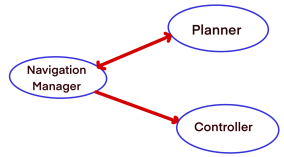
Image taken from *docs.ros.org*

1. in the graphical representation we can see
2. the goal service with a response as an aknowledgment
3. the Feedback topic
4. the result service

- A navigation request



- A navigation request

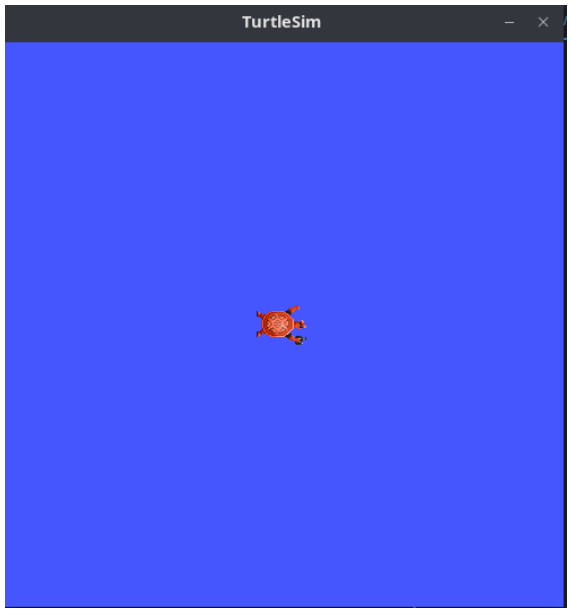


1. A perfect example of actions is a navigation request
2. The planner node gives the trajectories to follow to reach the goal
3. the navigation manager witch coordinates between the planer and the controller
4. the planner sets a goal and its notified periodically of the progress

Turtlesim and the ros command line tool

- A lightweight simulator for learning ROS2
- The ros2 command line tool is how the user manages, introspects, and interacts with a ROS System.

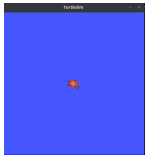
```
$ sudo apt install ros-jazzy-turtlesim
$ ros2 run turtlesim turtlesim_node
```



Turtlesim and the ros command line tool

- A lightweight simulator for learning ROS2
- The ros2 command line tool is how the user manages, introspects, and interacts with a ROS System.

```
$ sudo apt install ros-jazzy-turtlesim
$ ros2 run turtlesim turtlesim_node
```

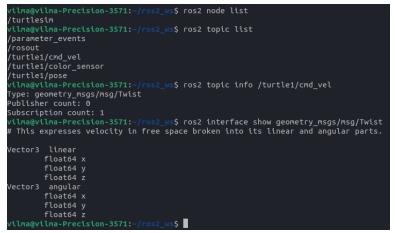


1. Ros provides us with a very light simulator for learning
2. but it doesnt come with the main installation so we need to install it
3. what the turtle does (like "draw" and "receive specific message types")
4. related to the distribution
5. Ros also provides us with a command line tool to interact and manage a Ros system
6. to run a node we type the command ros2 run the package and name of executable
7. we will discuss these elements in the following slides
8. we can see the window and the turtle robot in the middle
9. you can run the ros2 pkg executables turtlesim

Ros command line tool

```
vilma@vilma-Precision-3571:~/ros2_ws$ ros2 node list
/turtlesim
vilma@vilma-Precision-3571:~/ros2_ws$ ros2 topic list
/parameter_events
/rosout
/turtle1/cmd_vel
/turtle1/color_sensor
/turtle1/pose
vilma@vilma-Precision-3571:~/ros2_ws$ ros2 topic info /turtle1/cmd_vel
Type: geometry_msgs/msg/Twist
Publisher count: 0
Subscription count: 1
vilma@vilma-Precision-3571:~/ros2_ws$ ros2 interface show geometry_msgs/msg/Twist
# This expresses velocity in free space broken into its linear and angular parts.

Vector3  linear
        float64 x
        float64 y
        float64 z
Vector3  angular
        float64 x
        float64 y
        float64 z
vilma@vilma-Precision-3571:~/ros2_ws$
```

Ros command line tool

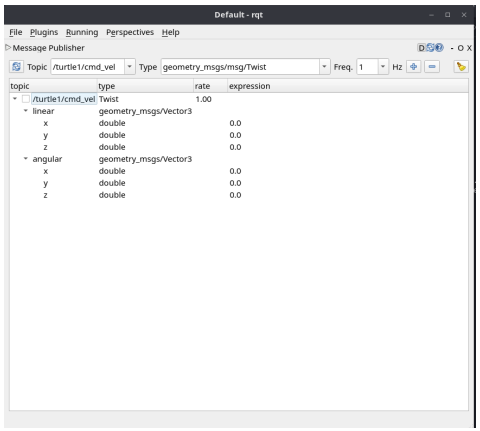


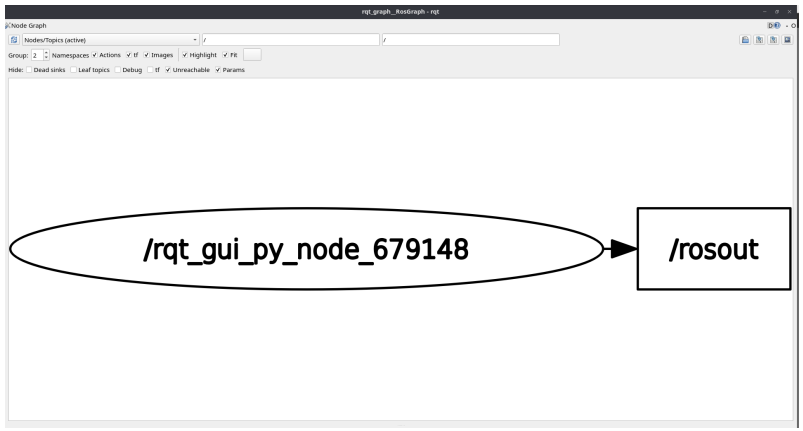
1. Lets take a closer look at the command line tool
2. the ros2 CLI is structured into commands and subcommands
3. "you can inspect both static and dynamic aspects of the system, like who's subscribed to what and message type definitions."
4. list the topic currently in use
5. get more info for the topic
6. show more about the message types
7. interfaces cover all three types of communication : topics, services, or actions.
8. in this example we can see the structure of the Twist message
9. 2 vectors representing the linear and angular velocity

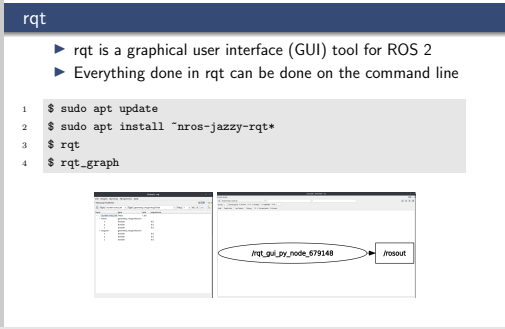
rqt

- rqt is a graphical user interface (GUI) tool for ROS 2
- Everything done in rqt can be done on the command line

```
1 $ sudo apt update
2 $ sudo apt install ~nros-jazzy-rqt*
3 $ rqt
4 $ rqt_graph
```







1. Ros also has a more user friendly graphical interface called rqt
2. we also need to install it
3. it provides the same capabilities as the command line tool
4. not so much used from the developers
5. but the graph generation part of the graphical interface is really quite used
6. it generates the graphical representation of the computational graph that we mentionned earlier

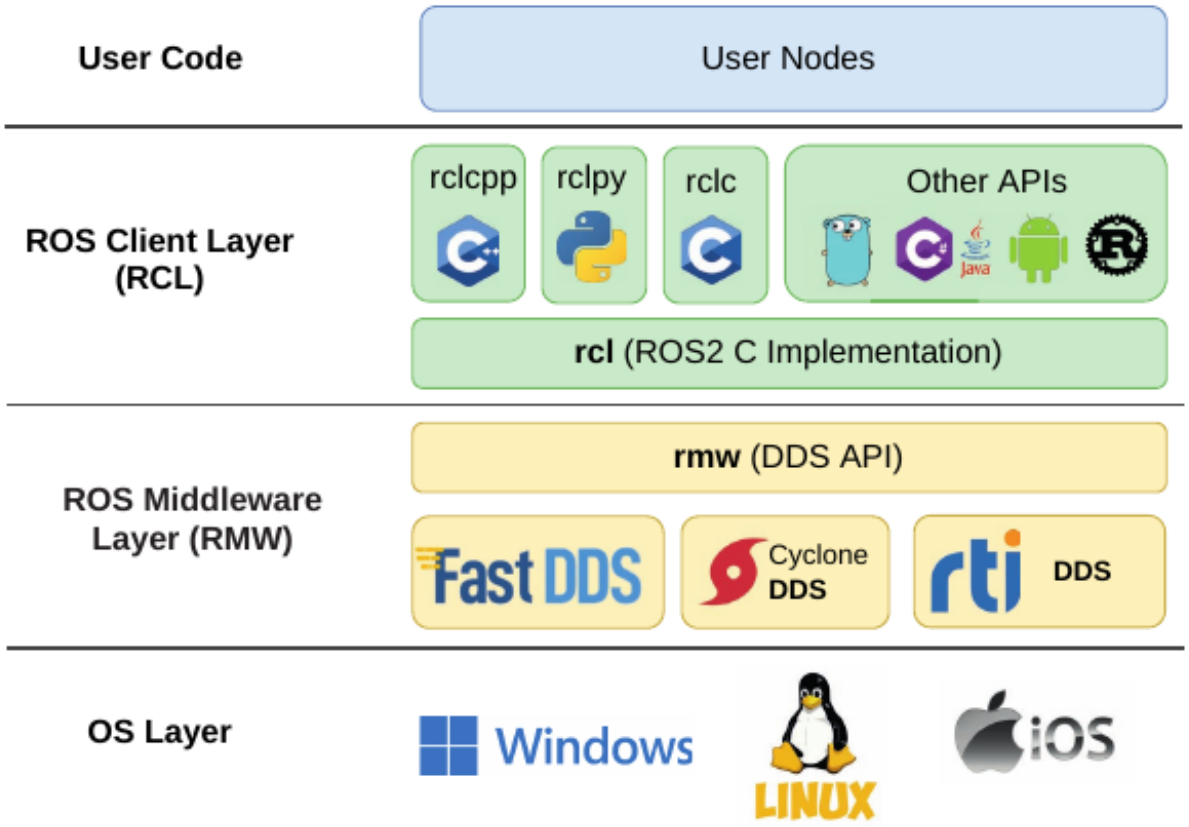


Image taken from the book *Introduction to Robot Programming With Ros2*

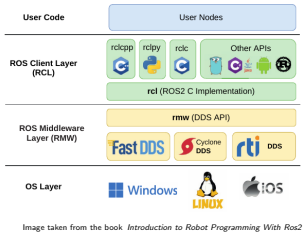
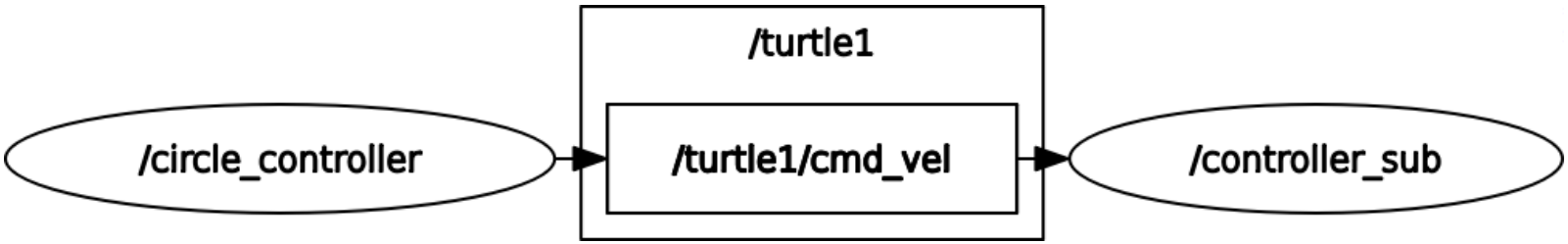


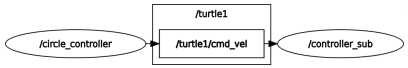
Image taken from the book *Introduction to Robot Programming With Ros2*

1. Ros has a layer structure
2. the lower layer deals with the problems of message transportation
3. while in Ros1 we had a core node that dealt with keeping track of messages publishers and subscriber and message trasportation
4. in ros2 they make use of a service provider Data distribution service which handles all these problems
5. ros has a middleware layer which allows to choose which DDS provider we want
6. we want see much difference expept if we are developing really time critical applications
7. then we have the main implementation of ros which uses c libraries

- The goal is to send velocity commands in order to move in a circle



- The goal is to send velocity commands in order to move in a circle



1. Lets look at an example
2. The goal of the program is to draw a circle
3. create a node that publish velocity commands so that the robot goes in a circle
4. and a subscriber node that in our case is only going to write the velocities on the terminal so we can just verify what values are received

- A package contains executables, libraries, or message definitions with a common purpose.
- colcon = collective contruction is a command line tool for building, testing, and using multiple software packages.

- A package contains executables, libraries, or message definitions with a common purpose.
- colcon = collective contruction is a command line tool for building, testing, and using multiple software packages.

1. the smallest unit of software development is a package
2. the package contains executables, libraries and message definitions with a common purpose
3. in order to manage packages we are going to use colcon
4. the name stands for collective construction and is a command line tool for building, testing and using multiple sofware packages

- Source the setup file or add the command to your shell startup script:

```
$ source /opt/ros/jazzy/setup.bash
$ echo "source /opt/ros/jazzy/setup.bash" >> ~/.bashrc
```

- Create a workspace and a package

```
1 $ mkdir -p ~/ros2_ws/src
2 $ cd ~/ros2_ws/src
3 $ ros2 pkg create --build-type ament_python --node-name \
4   circle_controller  p_controller --license Apache-2.0
5 $ colcon build --packages-select p_controller
6 $ source install/local_setup.bash
```

- Source the setup file or add the command to your shell startup script:

```
$ source /opt/ros/jazzy/setup.bash
$ echo "source /opt/ros/jazzy/setup.bash" >> ~/.bashrc
```

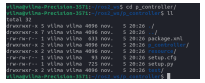
- Create a workspace and a package

```
1 $ mkdir -p ~/ros2_ws/src
2 $ cd ~/ros2_ws/src
3 $ ros2 pkg create --build-type ament_python --node-name \
4   circle_controller  p_controller --license Apache-2.0
5 $ colcon build --packages-select p_controller
6 $ source install/local_setup.bash
```

1. we start by activating the main ros installation in this case jazzy
2. we can also put the setup file in the startup script of the shell
3. this is nice if we dont plan to work with different distros
4. because then we risk to forget to source in each new terminal window
5. then we create a workspace - basically a folder
6. then a package with the command line tool lines 3 and 4
7. we specify which language we are going to use
8. the node name
9. also we can add other option like the license and dependancies etc
10. then we call colcon to build the package after writing our

```
vilma@vilma-Precision-3571:~/ros2_ws$ cd p_controller/
vilma@vilma-Precision-3571:~/ros2_ws/p_controller$ ll
total 32
drwxrwxr-x 5 vilma vilma 4096 nov.   5 20:26 ./
drwxrwxr-x 7 vilma vilma 4096 nov.   5 20:26 ../
-rw-rw-r-- 1 vilma vilma  633 nov.   5 20:26 package.xml
drwxrwxr-x 2 vilma vilma 4096 nov.   5 20:26 p_controller/
drwxrwxr-x 2 vilma vilma 4096 nov.   5 20:26 resource/
-rw-rw-r-- 1 vilma vilma   93 nov.   5 20:26 setup.cfg
-rw-rw-r-- 1 vilma vilma  725 nov.   5 20:26 setup.py
drwxrwxr-x 2 vilma vilma 4096 nov.   5 20:26 test/
vilma@vilma-Precision-3571:~/ros2_ws/p_controller$
```

- package.xml - file containing meta information about the package
- package_name - a directory with the same name as your package, used by ROS 2 tools to find your package, contains __init__.py
- resource/package_name marker file for the package
- setup.cfg - required when a package has executables, so ros2 run can find them
- setup.py - containing instructions for how to install the package



- package.xml - file containing meta information about the package
- package_name - a directory with the same name as your package, used by ROS 2 tools to find your package, contains __init__.py
- resource/package_name marker file for the package
- setup.cfg - required when a package has executables, so ros2 run can find them
- setup.py - containing instructions for how to install the package

1. Lets see the structure of the package that ros created for us
2. the important files are the
3. package.xml containing the meta information about the package
4. the setup.py containing instructions for how to install the packages
5. the folder with the same name as the package where we are going to put our python programs

Customize package.xml

- Description, License, Maintainer

```
<?xml version="1.0"?>
<?xml-model href="http://download.ros.org/schema/package_format3.xsd" schematypens="http://www.w3.org/2001/XMLSchema"?>
<package format="3">
  <name>p_controller</name>
  <version>0.0.0</version>
  <description>TODO: Package description</description>
  <maintainer email="vilma.muco@exail.com">vilma</maintainer>
  <license>TODO: License declaration</license>

  <test_depend>ament_copyright</test_depend>
  <test_depend>ament_flake8</test_depend>
  <test_depend>ament_pep257</test_depend>
  <test_depend>python3-pytest</test_depend>

  <export>
    <build_type>ament_python</build_type>
  </export>
</package>
```

Customize package.xml

- Description, License, Maintainer

1. the next step is to modify the package xml and add the lisence maintainer and description

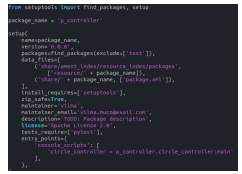
Customize setup.py

- Same description, maintainer and license fields as package.xml
- The version and name (package_name) also need to **match exactly**

```
from setuptools import find_packages, setup

package_name = 'p_controller'

setup(
    name=package_name,
    version='0.0.0',
    packages=find_packages(exclude=['test']),
    data_files=[
        ('share/ament_index/resource_index/packages',
            ['resource/' + package_name]),
        ('share/' + package_name, ['package.xml']),
    ],
    install_requires=['setuptools'],
    zip_safe=True,
    maintainer='vilma',
    maintainer_email='vilma.muco@exail.com',
    description='TODO: Package description',
    license='Apache License 2.0',
    tests_require=['pytest'],
    entry_points={
        'console_scripts': [
            'circle_controller = p_controller.circle_controller:main'
        ],
    },
)
```

Customize setup.py

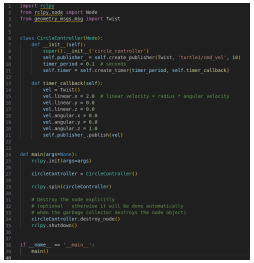
- Same description, maintainer and license fields as package.xml
- The version and name (package_name) also need to **match exactly**



1. we make the same modifications to the setup file
2. the description license and maintainer need to be the same in order to compile
3. The version and name also need to match exactly

Publisher python code

```python
import rclpy
from rclpy.node import Node
from geometry_msgs.msg import Twist


class CircleController(Node):
    def __init__(self):
        super().__init__('circle_controller')
        self.publisher_ = self.create_publisher(Twist, 'turtle1/cmd_vel', 10)
        timer_period = 0.1  # seconds
        self.timer = self.create_timer(timer_period, self.timer_callback)

    def timer_callback(self):
        vel = Twist()
        vel.linear.x = 2.0  # linear velocity = radius * angular velocity
        vel.linear.y = 0.0
        vel.linear.z = 0.0
        vel.angular.x = 0.0
        vel.angular.y = 0.0
        vel.angular.z = 1.0
        self.publisher_.publish(vel)


def main(args=None):
    rclpy.init(args=args)

    circleController = CircleController()

    rclpy.spin(circleController)

    # Destroy the node explicitly
    # (optional - otherwise it will be done automatically
    # when the garbage collector destroys the node object)
    circleController.destroy_node()
    rclpy.shutdown()


if __name__ == '__main__':
    main()

```



1. we have a general view of the publisher python code
2. we can see three blocks of code
3. the imports block
4. the class declaration
5. the main function

Code Analysis

```python
import rclpy
from rclpy.node import Node
from geometry_msgs.msg import Twist


class CircleController(Node):
    def __init__(self):
        super().__init__('circle_controller')
        self.publisher_ = self.create_publisher(Twist, 'turtle1/cmd_vel', 10)
        timer_period = 0.1  # seconds
        self.timer = self.create_timer(timer_period, self.timer_callback)

    def timer_callback(self):
        vel = Twist()
        vel.linear.x = 2.0  # linear velocity = radius * angular velocity
        vel.linear.y = 0.0
        vel.linear.z = 0.0
        vel.angular.x = 0.0
        vel.angular.y = 0.0
        vel.angular.z = 1.0
        self.publisher_.publish(vel)
```

Code Analysis

```python
import rclpy
from rclpy.node import Node
from geometry_msgs.msg import Twist


class CircleController(Node):
    def __init__(self):
        super().__init__('circle_controller')
        self.publisher_ = self.create_publisher(Twist, 'turtle1/cmd_vel', 10)
        timer_period = 0.1  # seconds
        self.timer = self.create_timer(timer_period, self.timer_callback)

    def timer_callback(self):
        vel = Twist()
        vel.linear.x = 2.0  # linear velocity = radius * angular velocity
        vel.linear.y = 0.0
        vel.linear.z = 0.0
        vel.angular.x = 0.0
        vel.angular.y = 0.0
        vel.angular.z = 1.0
        self.publisher_.publish(vel)
```

1. python client library
2. the Node class from which our node is going to inherit
3. The Twist message for our topic
4. the base constractor which takes as argument the node name
5. next we create a publisher
6. argument of the publisher
7. Queue size is a required QoS setting that limits the amount of queued messages if a subscriber is not receiving them fast enough
8. then we create a timer in order to have an regular publishing rate
9. The timer also defines whitch function is going to call back

Code Analysis

```
1  def main(args=None):
2      rclpy.init(args=args)
3
4      circleController = CircleController()
5
6      rclpy.spin(circleController)
7
8      # Destroy the node explicitly
9      # (optional - otherwise it will be done automatically
10     # when the garbage collector destroys the node object)
11     circleController.destroy_node()
12     rclpy.shutdown()
```

Code Analysis

```python
def main(args=None):
    rclpy.init(args=args)

    circleController = CircleController()

    rclpy.spin(circleController)

    # Destroy the node explicitly
    # (optional - otherwise it will be done automatically
    # when the garbage collector destroys the node object)
    circleController.destroy_node()
    rclpy.shutdown()
```

1. Lastly, the main function is defined.
2. First the rclpy library is initialized
3. then the node is created
4. and then it spins the node so its callbacks are called

Code Analysis

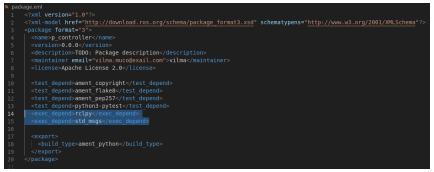
- Add dependencies

package.xml

```
1  <?xml version="1.0"?>
2  <?xml-model href="http://download.ros.org/schema/package_format3.xsd" schematypens="http://www.w3.org/2001/XMLSchema"?>
3  <package format="3">
4    <name>p_controller</name>
5    <version>0.0.0</version>
6    <description>TODO: Package description</description>
7    <maintainer email="vilma.muco@exail.com">vilma</maintainer>
8    <license>Apache License 2.0</license>
9
10   <test_depend>ament_copyright</test_depend>
11   <test_depend>ament_flake8</test_depend>
12   <test_depend>ament_pep257</test_depend>
13   <test_depend>python3-pytest</test_depend>
14   <exec_depend>rclpy</exec_depend>
15   <exec_depend>std_msgs</exec_depend>
16
17   <export>
18     <build_type>ament_python</build_type>
19   </export>
20 </package>
```

Code Analysis

- Add dependencies



1. Then we add the dependancies
2. The packages that we used and our program is dependant on at the package xml file
3. in this case the rclpy and std_msgs

Code Analysis

- Add entry point

setup.py > ...

```python
from setuptools import find_packages, setup

package_name = 'p_controller'

setup(
    name=package_name,
    version='0.0.0',
    packages=find_packages(exclude=['test']),
    data_files=[
        ('share/ament_index/resource_index/packages',
            ['resource/' + package_name]),
        ('share/' + package_name, ['package.xml']),
    ],
    install_requires=['setuptools'],
    zip_safe=True,
    maintainer='vilma',
    maintainer_email='vilma.muco@exail.com',
    description='TODO: Package description',
    license='Apache License 2.0',
    tests_require=['pytest'],
    entry_points={
        'console_scripts': [
            'circle_controller = p_controller.circle_controller:main'
        ],
    },
)
```

- Add entry point

1. Then we add the executable on the setup file
2. This normaly is already created but the ros2 pkg create when we gave the node name
3. we are going to specify the executable name and the fuction to be called upon the run command
4. line 23

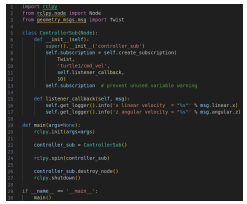
Create a subscriber node

- Create the controller_subscriber.py file in the same folder as the circle_controller.py

```python
import rclpy
from rclpy.node import Node
from geometry_msgs.msg import Twist

class ControllerSub(Node):
    def __init__(self):
        super().__init__('controller_sub')
        self.subscription = self.create_subscription(
            Twist,
            'turtle1/cmd_vel',
            self.listener_callback,
            10)
        self.subscription  # prevent unused variable warning

    def listener_callback(self, msg):
        self.get_logger().info('x linear velocity  = "%s"' % msg.linear.x)
        self.get_logger().info('z angular velocity = "%s"' % msg.angular.z)

def main(args=None):
    rclpy.init(args=args)

    controller_sub = ControllerSub()

    rclpy.spin(controller_sub)

    controller_sub.destroy_node()
    rclpy.shutdown()

if __name__ == '__main__':
    main()
```

Create a subscriber node

- Create the controller_subscriber.py file in the same folder as the circle_controller.py



1. Create a subscriber file in the same forder as the publisher python file
2. we and see again the three blocks
3. the import block
4. class declaration
5. the main function

Subscriber Code Analysis

```
1  import rclpy
2  from rclpy.node import Node
3  from geometry_msgs.msg import Twist
4
5  class ControllerSub(Node):
6      def __init__(self):
7          super().__init__('controller_sub')
8          self.subscription = self.create_subscription(
9              Twist,
10             'turtle1/cmd_vel',
11             self.listener_callback,
12             10)
13
14     def listener_callback(self, msg):
15         self.get_logger().info('x linear velocity  = "%s"' % msg.linear.x)
16         self.get_logger().info('z angular velocity = "%s"' % msg.angular.z)
17
```

Subscriber Code Analysis

```python
import rclpy
from rclpy.node import Node
from geometry_msgs.msg import Twist

class ControllerSub(Node):
    def __init__(self):
        super().__init__('controller_sub')
        self.subscription = self.create_subscription(
            Twist,
            'turtle1/cmd_vel',
            self.listener_callback,
            10)

    def listener_callback(self, msg):
        self.get_logger().info('x linear velocity  = "%s"' % msg.linear.x)
        self.get_logger().info('z angular velocity = "%s"' % msg.angular.z)

```

1. The difference here is the creation of the subscriber which takes as arguments the Type of the message
2. the name of the topic to subscribe to
3. the function that is going to be called to handle the message
4. The subscriber's constructor and callback don't include any timer definition, because it doesn't need one.
5. Its callback gets called as soon as it receives a message.
6. in the callback function we print the velocities at the console

Subscriber Code Analysis

```
1  def main(args=None):
2      rclpy.init(args=args)
3      controller_sub = ControllerSub()
4
5      rclpy.spin(controller_sub)
6
7      controller_sub.destroy_node()
8      rclpy.shutdown()
9
10 if __name__ == '__main__':
11     main()
```

Subscriber Code Analysis

```python
def main(args=None):
    rclpy.init(args=args)
    controller_sub = ControllerSub()

    rclpy.spin(controller_sub)

    controller_sub.destroy_node()
    rclpy.shutdown()

if __name__ == '__main__':
    main()
```

1. The main block is exactly the same as on the publisher file
2. First the rclpy library is initialized, then the node is created, and then it spins the node so its callbacks are called.

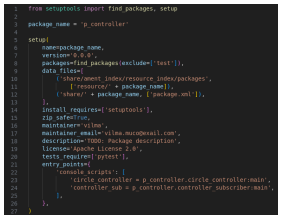
Subscriber Code Analysis

- Add dependencies
- Add entry point (line 24)

```
1  from setuptools import find_packages, setup
2
3  package_name = 'p_controller'
4
5  setup(
6      name=package_name,
7      version='0.0.0',
8      packages=find_packages(exclude=['test']),
9      data_files=[
10         ('share/ament_index/resource_index/packages',
11             ['resource/' + package_name]),
12         ('share/' + package_name, ['package.xml']),
13     ],
14     install_requires=['setuptools'],
15     zip_safe=True,
16     maintainer='vilma',
17     maintainer_email='vilma.muco@exail.com',
18     description='TODO: Package description',
19     license='Apache License 2.0',
20     tests_require=['pytest'],
21     entry_points={
22         'console_scripts': [
23             'circle_controller = p_controller.circle_controller:main',
24             'controller_sub = p_controller.controller_subscriber:main',
25         ],
26     },
27 )
```

Subscriber Code Analysis

- Add dependencies
- Add entry point (line 24)



1. The dependancies are already added because we used the same packages as the publisher so we can actually skip this step
2. then we add a new line at the entry points of the setup file defining the new executable and the function to be called when we run it

Build and run

- Run in different terminals the publisher and subscriber

```
1 $ colcon build --packages-select p_controller
2 $ source install/setup.bash
3 $ ros2 run p_controller circle_controller
4 $ ros2 run p_controller controller_sub
```

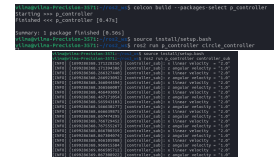
```
vilma@vilma-Precision-3571:~/ros2_ws$ colcon build --packages-select p_controller
Starting >>> p_controller
Finished <<< p_controller [0.47s]

Summary: 1 package finished [0.56s]
vilma@vilma-Precision-3571:~/ros2_ws$ source install/setup.bash
vilma@vilma-Precision-3571:~/ros2_ws$ ros2 run p_controller circle_controller
```

```
vilma@vilma-Precision-3571:~/ros2_ws$ source install/setup.bash
vilma@vilma-Precision-3571:~/ros2_ws$ ros2 run p_controller controller_sub
[INFO] [1699286368.171228256] [controller_sub]: x linear velocity  = "2.0"
[INFO] [1699286368.171394386] [controller_sub]: z angular velocity = "1.0"
[INFO] [1699286368.266327440] [controller_sub]: x linear velocity  = "2.0"
[INFO] [1699286368.266923092] [controller_sub]: z angular velocity = "1.0"
[INFO] [1699286368.366044594] [controller_sub]: x linear velocity  = "2.0"
[INFO] [1699286368.366566007] [controller_sub]: z angular velocity = "1.0"
[INFO] [1699286368.466493093] [controller_sub]: x linear velocity  = "2.0"
[INFO] [1699286368.467171054] [controller_sub]: z angular velocity = "1.0"
[INFO] [1699286368.565943103] [controller_sub]: x linear velocity  = "2.0"
[INFO] [1699286368.566630277] [controller_sub]: z angular velocity = "1.0"
[INFO] [1699286368.666639875] [controller_sub]: x linear velocity  = "2.0"
[INFO] [1699286368.667474295] [controller_sub]: z angular velocity = "1.0"
[INFO] [1699286368.766729452] [controller_sub]: x linear velocity  = "2.0"
[INFO] [1699286368.767555267] [controller_sub]: z angular velocity = "1.0"
[INFO] [1699286368.866700359] [controller_sub]: x linear velocity  = "2.0"
[INFO] [1699286368.867304074] [controller_sub]: z angular velocity = "1.0"
[INFO] [1699286368.966105988] [controller_sub]: x linear velocity  = "2.0"
[INFO] [1699286368.966915584] [controller_sub]: z angular velocity = "1.0"
[INFO] [1699286369.066505712] [controller_sub]: x linear velocity  = "2.0"
[INFO] [1699286369.067308922] [controller_sub]: z angular velocity = "1.0"
```

Build and run

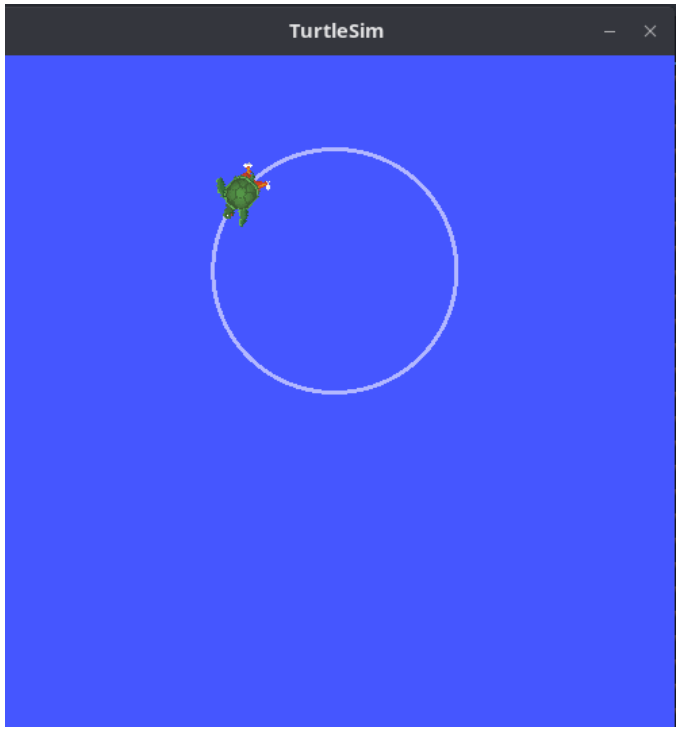
- Run in different terminals the publisher and subscriber

```
$ colcon build --packages-select p_controller
$ source install/setup.bash
$ ros2 run p_controller circle_controller
$ ros2 run p_controller controller_sub
```

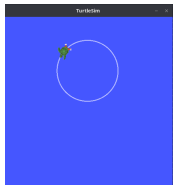


1. now we can build and run the program
2. so in two different terminals
3. we run the colcon build command
4. dont forget to source the setup of the workspace
5. then run the executables
6. we can see that on the second terminal that we are receiving the messages

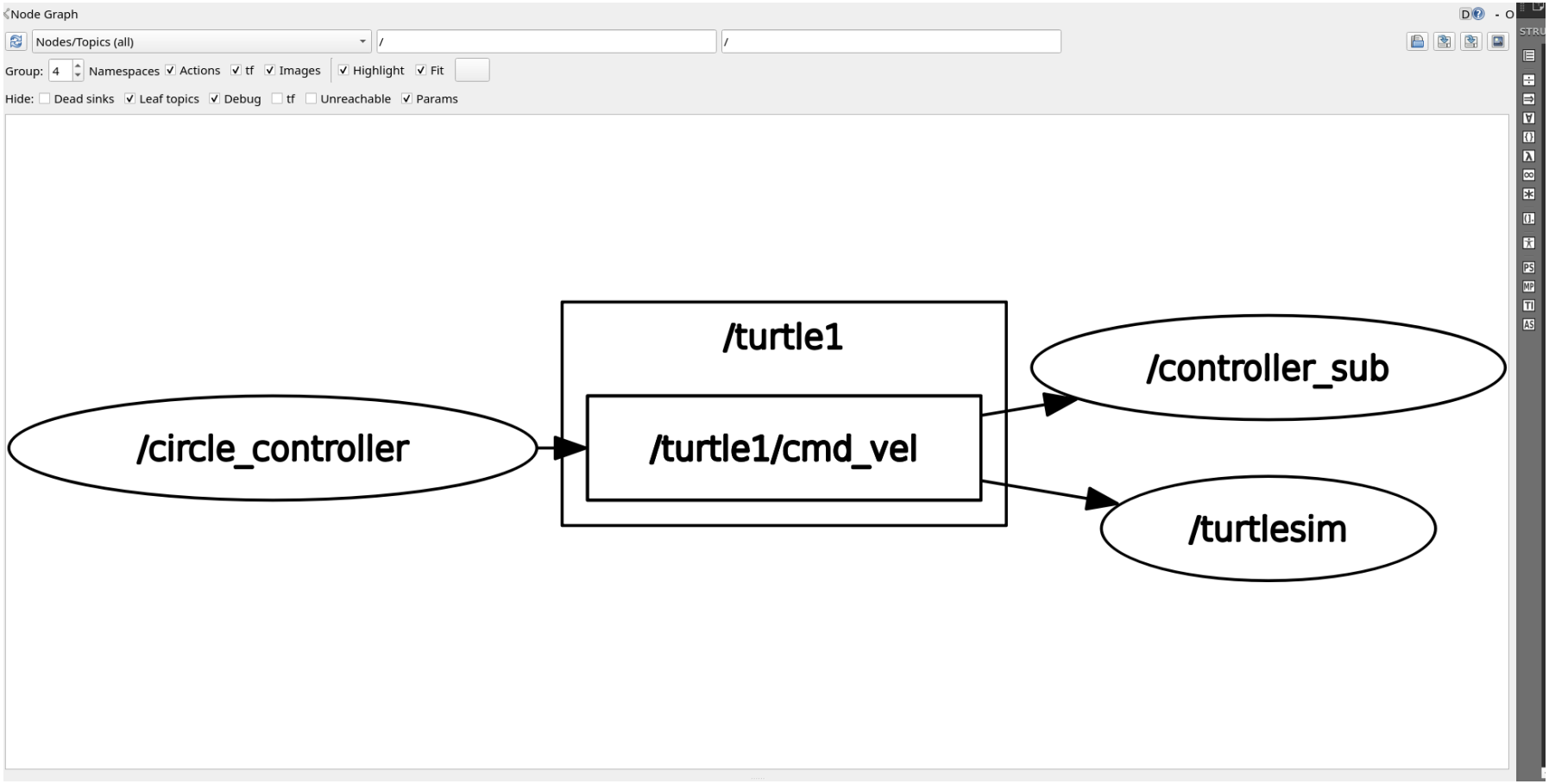
- Run Turtlesim and rqt_graph

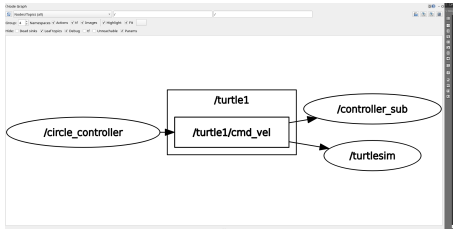


- Run Turtlesim and rqt_graph



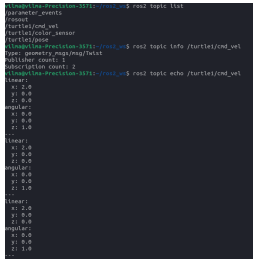
1. now if we run the turtle sim in a third terminal
2. and because we used the same topic also used by turtlesim
3. we see the turle moving in a circle!


Node Graph
Nodes/Topics (all)
Group: 4 Namespaces Actions tf Images Highlight Fit
Hide: Dead sinks Leaf topics Debug tf Unreachable Params
/turtle1
/circle_controller
/turtle1/cmd_vel
/controller_sub
/turtlesim



1. if we launch rqt graph we can see that the turtle sim and the subscriber node are both subscribed to the same topic

```
vilma@vilma-Precision-3571:~/ros2_ws$ ros2 topic list
/parameter_events
/rosout
/turtle1/cmd_vel
/turtle1/color_sensor
/turtle1/pose
vilma@vilma-Precision-3571:~/ros2_ws$ ros2 topic info /turtle1/cmd_vel
Type: geometry_msgs/msg/Twist
Publisher count: 1
Subscription count: 2
vilma@vilma-Precision-3571:~/ros2_ws$ ros2 topic echo /turtle1/cmd_vel
linear:
  x: 2.0
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 1.0
---
linear:
  x: 2.0
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 1.0
---
linear:
  x: 2.0
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 1.0
---
```



1. we can use the command line to check as well what is happening
2. we can see the list of the topics active
3. the info how many publishers and subscribers for a specific topic
4. ”of course we wouldn’t in the real world write a node just to echo its input, ros2 provides commands for that”
5. and we can listen the mesages currently being published on a topic with the echo command

- Launch files allow you to start up and configure a number of executables containing ROS 2 nodes simultaneously.
- You can write the files in different languages:
  - Python
  - XML
  - YAML

- Launch files allow you to start up and configure a number of executables containing ROS 2 nodes simultaneously.
- You can write the files in different languages:
  - Python
  - XML
  - YAML

1. As we saw even for a small example we needed 3 terminals
2. so your can imagine in actual systems with hundreds and thousand of nodes....
3. especially if the nodes need configuration parameters like for a gps receiver the ip and port number
4. finding the executables and then passing parameters it will be a nightmare
5. So ros comes to our rescue by providing us with the possibility to use launch files
6. you can write launch files in three ways
7. in python
8. in xml
9. in yaml
10. here we are going to use the python file

- A program that contains a LaunchDescription object
- The LaunchDescription object contains actions:
  - Node action - to run a program
  - IncludeLaunchDescription
  - DeclareLaunchArguments
  - SetEnvironmentVariable
  - ...

- A program that contains a LaunchDescription object
- The LaunchDescription object contains actions:
  - Node action - to run a program
  - IncludeLaunchDescription
  - DeclareLaunchArguments
  - SetEnvironmentVariable
  - ...

1. A python launch file is a python program that contains a LaunchDescription object
2. the LaunchDescription object contains actions that can run a node
3. include another launch description file
4. can be really useful to separate then according to functionality example navigation nodes, communication driver nodes etc
5. it can declare arguments and pass them to the nodes
6. set environment variables etc

- Create a new directory to store your launch files:

```
$ mkdir launch
```

- Let's put together a ROS 2 launch file using the turtlesim package and the circle_controller publisher.

- Create a new directory to store your launch files:

```
$ mkdir launch
```

- Let's put together a ROS 2 launch file using the turtlesim package and the circle_controller publisher.

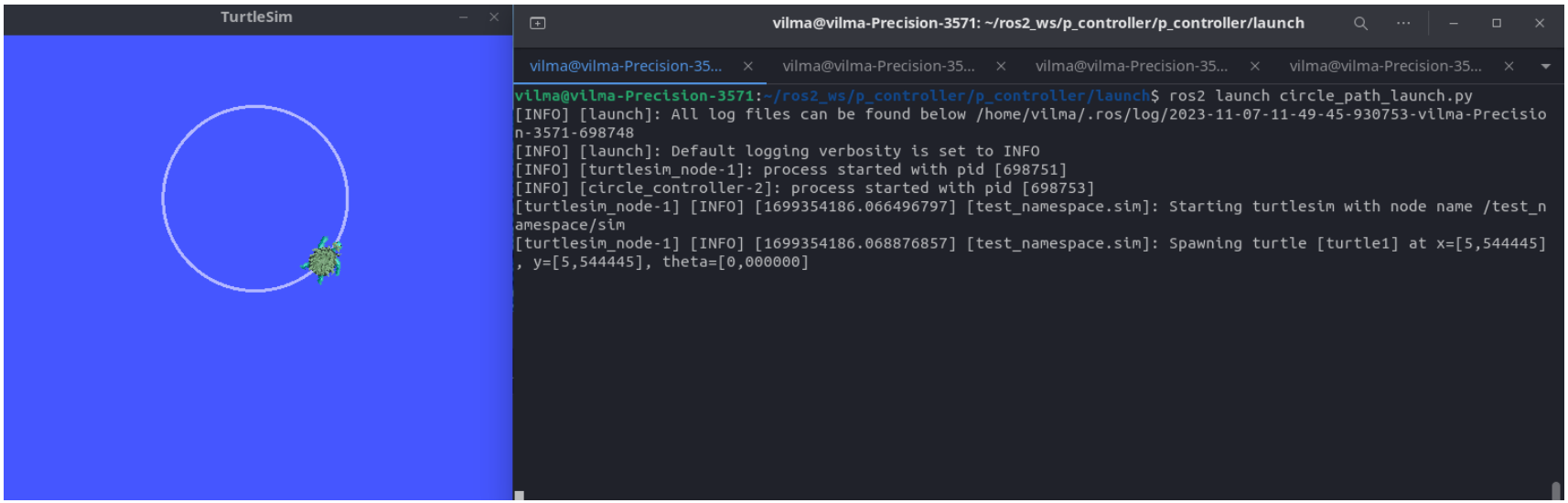
1. It is common practice to put the launch files in a launch folder so first thing we do is create a launch file in our package
2. and lets put together in a launch file the publisher node and a turtlesim node

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


def generate_launch_description():
    return LaunchDescription([
        Node(
            package='turtlesim',
            namespace='test_namespace',
            executable='turtlesim_node',
            name='sim'
        ),
        Node(
            package='p_controller',
            namespace='test_namespace',
            executable='circle_controller',
            name='controller'
        )
    ])

```

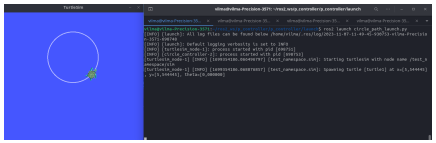
```python
from launch import LaunchDescription
from launch_ros.actions import Node


def generate_launch_description():
    return LaunchDescription([
        Node(
            package='turtlesim',
            namespace='test_namespace',
            executable='turtlesim_node',
            name='sim'
        ),
        Node(
            package='p_controller',
            namespace='test_namespace',
            executable='circle_controller',
            name='controller'
        )
    ])

```

1. for each node we specify the package
2. the namespace
3. the name

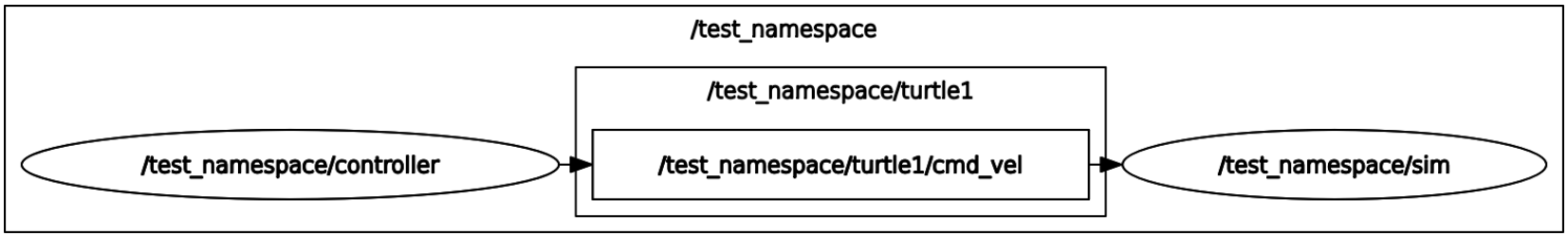
```
$ cd launch
$ ros2 launch circle_path_launch.py
```

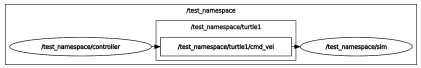


```
$ cd launch
$ ros2 launch circle_path_launch.py
```



1. we can execute the launch file with the command ros2 launch and the path to the file
2. and we see a turtlesim window pop up and turtle moving in circle


/test_namespace
/test_namespace/turtle1
/test_namespace/controller
/test_namespace/turtle1/cmd_vel
/test_namespace/sim



1. if we run the rqt graph we can see also the added namespace in front of the resources nodes and topics

- Install Jazzy Jalisco

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

- Write a python publisher node that sends velocity commands to move the turtle in a square (Assume that we do not have any message loss)

Setup instructions and slides:
https://github.com/vilmamuco/ros2-exercises

- Python tutorial: https://www.w3schools.com/python/
- Linux tutorial: Bash 101 - Working at the CLI

Thank you!

Questions?

Temporary page!

LaTeX was unable to guess the total number of pages correctly. A
there was some unprocessed data that should have been added
the final page this extra page has been added to receive it.
If you rerun the document (without altering it) this surplus page
will go away, because LaTeX now knows how many pages to expe
for this document.