

Packages and launch files in 2

Olivier Kermorgant

ANF ROS2

Regroup commands that should work together

- Run several nodes at the same time
- Remap topics (hard-coded in node code source)
- Run nodes inside a namespace
- Set / load some parameters
- Include other launch files

`https://github.com/oKermorgant/anf_launch`

- `apt install ros-${ROS_DISTRO}-slider-publisher`
- `apt install ros-${ROS_DISTRO}-simple-launch`

Regroup commands that should work together

- Run several nodes at the same time
- Remap topics (hard-coded in node code source)
- Run nodes inside a namespace
- Set / load some parameters
- Include other launch files

`https://github.com/oKermorgant/anf_launch`

- `apt install ros-${ROS_DISTRO}-slider-publisher`
- `apt install ros-${ROS_DISTRO}-simple-launch`

Regroup commands that should work together

- Run several nodes at the same time
- Remap topics (hard-coded in node code source)
- Run nodes inside a namespace
- Set / load some parameters
- Include other launch files

`https://github.com/oKermorgant/anf_launch`

• `apt install ros-${ROS_DISTRO}-slider-publisher`

• `apt install ros-${ROS_DISTRO}-simple-launch`

Regroup commands that should work together

- Run several nodes at the same time
- Remap topics (hard-coded in node code source)
- Run nodes inside a namespace
- Set / load some parameters
- Include other launch files

`https://github.com/oKermorgant/anf_launch`

- `apt install ros-${ROS_DISTRO}-slider-publisher`
- `apt install ros-${ROS_DISTRO}-simple-launch`

Regroup commands that should work together

- Run several nodes at the same time
- Remap topics (hard-coded in node code source)
- Run nodes inside a namespace
- Set / load some parameters
- Include other launch files

`https://github.com/oKermorgant/anf_launch`

- `apt install ros-${ROS_DISTRO}-slider-publisher`
- `apt install ros-${ROS_DISTRO}-simple-launch`

Regroup commands that should work together

- Run several nodes at the same time
- Remap topics (hard-coded in node code source)
- Run nodes inside a namespace
- Set / load some parameters
- Include other launch files

`https://github.com/oKermorgant/anf_launch`

• `apt install ros-${ROS_DISTRO}-slider-publisher`

• `apt install ros-${ROS_DISTRO}-simple-launch`

Regroup commands that should work together

- Run several nodes at the same time
- Remap topics (hard-coded in node code source)
- Run nodes inside a namespace
- Set / load some parameters
- Include other launch files



https://github.com/oKermorgant/anf_launch

- `apt install ros-${ROS_DISTRO}-slider-publisher`
- `apt install ros-${ROS_DISTRO}-simple-launch`

History of distributions - Long Term Support are what you want

First
commit



In 2017: 200000 commits made by more than 2800 users
More than 2000 forks of `rostdistro` from package developpers

Running two nodes that should communicate

```
# publishes on /setpoint by default, needs a command line argument
ros2 run slider_publisher slider_publisher

# listens to /joint_setpoint by default
# needs a 'joint_name' parameter
ros2 run move_joint move_joint
```



/slider_publisher

/move_joint

```
> ros2 wtf --report
  TOPIC LIST
topic : /joint_setpoint
publisher count : 0
subscriber count : 1
topic : /setpoint
publisher count : 1
subscriber count : 0
```

Running two nodes that should communicate

```
# publishes on /setpoint by default, needs a command line argument
ros2 run slider_publisher slider_publisher

# listens to /joint_setpoint by default
# needs a 'joint_name' parameter
ros2 run move_joint move_joint
```



/slider_publisher

/move_joint

```
> ros2 wtf --report
  TOPIC LIST
topic : /joint_setpoint
publisher count : 0
subscriber count : 1
topic : /setpoint
publisher count : 1
subscriber count : 0
```

Remappings are for nodes not made to work together

```
ros2 run slider_publisher slider_publisher ./slider_config.yaml

ros2 run move_joint move_joint --ros-args -r /joint_setpoint:=/setpoint -p joint_name:=right_e0
```

Running two nodes that should communicate

```
# publishes on /setpoint by default, needs a command line argument
ros2 run slider_publisher slider_publisher

# listens to /joint_setpoint by default
# needs a 'joint_name' parameter
ros2 run move_joint move_joint
```

/slider_publisher

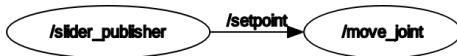
/move_joint

```
> ros2 wtf --report
  TOPIC LIST
topic : /joint_setpoint
publisher count : 0
subscriber count : 1
topic : /setpoint
publisher count : 1
subscriber count : 0
```

Remappings are for nodes not made to work together

```
ros2 run slider_publisher slider_publisher ./slider_config.yaml

ros2 run move_joint move_joint --ros-args -r /joint_setpoint:=/setpoint -p joint_name:=right_e0
```



Any file in ROS belongs to a given package

- Atomic way to share and identify code
- Can be CMake-based or pure Python

A package is identified by its `package.xml` file

- Give the name + dependencies (other ROS packages or other libraries)

Any file in ROS belongs to a given package

- Atomic way to share and identify code
- Can be CMake-based or pure Python

A package is identified by its `package.xml` file

- Give the name + dependencies (other ROS packages or other libraries)

```
1 <?xml version="1.0"?>
2 <package format="3">
3   <name>simulation_2d</name>
4   <version>2.0.0</version>
5   <description>The simulation2D package</description>
6   <maintainer email="olivier.kermorgant@ec-nantes.fr">Olivier Kermorgant</maintainer>
7
8   <license>MIT</license>
9   <buildtool_depend>ament_cmake</buildtool_depend>
10
11   <depend>geometry_msgs</depend>
12   <depend>roscpp</depend>
13   <depend>sensor_msgs</depend>
14   <depend>urdfdom</depend>
15
16   <export>
17     <build_type>ament_cmake</build_type>
18   </export>
19 </package>
```

A package may contain:

- C++ code → `include/` `src/`
- Python code → `scripts/`
- robot descriptions → `urdf/` `meshes/`
- launch files → `launch/`
- custom messages → `msg/` `srv/`
- actually any file

Package name is used:

- to run its nodes `ros2 run pkg node`
- as the namespace for its custom messages
- to find any file inside
- as a dependency for another package

ROS packages - what's inside?

A package may contain:

- C++ code → `include/` `src/`
- Python code → `scripts/`
- robot descriptions → `urdf/` `meshes/`
- launch files → `launch/`
- custom messages → `msg/` `srv/`
- actually any file

Package name is used:

- to run its nodes `ros2 run pkg node`
- as the namespace for its custom messages
- to find any file inside
- as a dependency for another package

> Home > code > ros2 > src > **ros2_nav_tutorial**

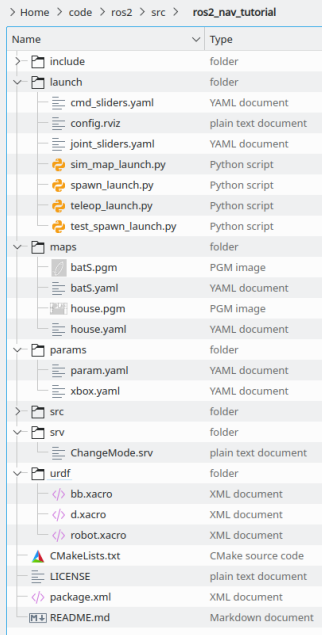
Name	Type
> include	folder
> launch	folder
cmd_sliders.yaml	YAML document
config.rviz	plain text document
joint_sliders.yaml	YAML document
sim_map_launch.py	Python script
spawn_launch.py	Python script
teleop_launch.py	Python script
test_spawn_launch.py	Python script
> maps	folder
bat5.pgm	PGM image
bat5.yaml	YAML document
house.pgm	PGM image
house.yaml	YAML document
> params	folder
param.yaml	YAML document
xbox.yaml	YAML document
> src	folder
> srv	folder
ChangeMode.srv	plain text document
> urdf	folder
bb.xacro	XML document
d.xacro	XML document
robot.xacro	XML document
CMakeLists.txt	CMake source code
LICENSE	plain text document
package.xml	XML document
README.md	Markdown document

A package may contain:

- C++ code → `include/` `src/`
- Python code → `scripts/`
- robot descriptions → `urdf/` `meshes/`
- launch files → `launch/`
- custom messages → `msg/` `srv/`
- actually any file

Package name is used:

- to run its nodes `ros2 run pkg node`
- as the namespace for its custom messages
- to find any file inside
- as a dependency for another package



The screenshot shows a file explorer window with the path `> Home > code > ros2 > src > ros2_nav_tutorial`. The file list is as follows:

Name	Type
<code>include</code>	folder
<code>launch</code>	folder
<code>cmd_sliders.yaml</code>	YAML document
<code>config.rviz</code>	plain text document
<code>joint_sliders.yaml</code>	YAML document
<code>sim_map_launch.py</code>	Python script
<code>spawn_launch.py</code>	Python script
<code>teleop_launch.py</code>	Python script
<code>test_spawn_launch.py</code>	Python script
<code>maps</code>	folder
<code>bat5.pgm</code>	PGM image
<code>bat5.yaml</code>	YAML document
<code>house.pgm</code>	PGM image
<code>house.yaml</code>	YAML document
<code>params</code>	folder
<code>param.yaml</code>	YAML document
<code>xbox.yaml</code>	YAML document
<code>src</code>	folder
<code>srv</code>	folder
<code>ChangeMode.srv</code>	plain text document
<code>urdf</code>	folder
<code>bb.xacro</code>	XML document
<code>d.xacro</code>	XML document
<code>robot.xacro</code>	XML document
<code>CMakeLists.txt</code>	CMake source code
<code>LICENSE</code>	plain text document
<code>package.xml</code>	XML document
<code>README.md</code>	Markdown document

ROS launch files



- Forward launch parameters to nodes
- Run commands, get their output
- Conditional or namespaced groups

ROS 2:

- More or less same as ROS 1
- Syntax slightly different
- No condition groups, no command output (xacro)
- De-facto standard, same capabilities as in ROS 1
- New features:
 - Composition
 - Adapting parameters from a YAML file
 - Doing pretty much anything you can do in Python
- Used in all tutorials / popular packages

ROS launch files



- Forward launch parameters to nodes
- Run commands, get their output
- Conditional or namespaced groups

ROS 2:



- More or less same as ROS 1
 - Syntax slightly different
 - No condition groups, no command output (xacro)
-
- De-facto standard, same capabilities as in ROS 1
 - New features:
 - composition
 - adapting parameters from a YAML file
 - actually anything you can do in Python
 - Used in all tutorials / popular packages

ROS launch files



- Forward launch parameters to nodes
- Run commands, get their output
- Conditional or namespaced groups

ROS 2:



- More or less same as ROS 1
- Syntax slightly different
- No condition groups, no command output (xacro)



- De-facto standard, same capabilities as in ROS 1
- New features:
 - composition
 - adapting parameters from a YAML file
 - actually anything you can do in Python
- Used in all tutorials / popular packages

XML launch: ROS 1 vs ROS 2

ROS

```
1 <?xml version="1.0"?>
2 <launch>
3
4 <node name="rviz" pkg="rviz" type="rviz" respawn="true" output="screen"
5     args="-d $(find my_package)/launch/config.rviz"/>
6
7 <!-- Namespacing / xacro-->
8 <group ns="bb8">
9     <param name="robot_description" command="$(find xacro)/xacro $(find my_package)/urdf/bb8.xacro"/>
10    <node name="robot_state_publisher" pkg="robot_state_publisher" type="robot_state_publisher"/>
11 </group>
12 </launch>
```

2

```
1 <?xml version="1.0"?>
2 <launch>
3
4 <node name="rviz2" pkg="rviz2" exec="rviz2" respawn="true" output="screen"
5     args="-d $(find-pkg-share my_package)/launch/config.rviz"/>
6
7 <!-- no xacro... have to use URDF -->
8 <group>
9     <push-ros-namespace namespace="bb8"/>
10    <node name="robot_state_publisher" pkg="robot_state_publisher" exec="robot_state_publisher"
11        args="$(find-pkg-share my_package)/urdf/bb8.urdf"/>
12 </group>
13 </launch>
```

Python or XML launch files: running RViz

```
1 <?xml version="1.0"?>
2 <launch>
3 <node name="rviz2" pkg="rviz2" exec="rviz2" respawn="true" output="screen"
4     args="-d $(find-pkg-share my_package)/launch/config.rviz"/>
5 </launch>
```



```
import os
from ament_index_python.packages import get_package_share_directory
from launch import LaunchDescription
from launch_ros.actions import Node

def generate_launch_description():
    rviz_config_dir = os.path.join(
        get_package_share_directory('my_package'),
        'launch',
        'config.rviz')

    return LaunchDescription([
        Node(
            package='rviz2',
            executable='rviz2',
            name='rviz2',
            arguments=['-d', rviz_config_dir],
            output='screen', respawn='true'),
    ])
```



Basic launch file that runs 2 nodes without remapping

```
ros2 run slider_publisher slider_publisher # publishes on /setpoint
ros2 run move_joint move_joint # listens to /joint_setpoint
```

```
from simple_launch import SimpleLauncher
```

```
def generate_launch_description():
```

```
    sl = SimpleLauncher()
```

```
    sl.node('slider_publisher', 'slider_publisher')
```

```
    sl.node('move_joint', 'move_joint')
```

```
    return sl.launch_description()
```

/slider_publisher

/move_joint

Basic launch file that runs 2 nodes with remapping

```
ros2 run slider_publisher slider_publisher path/to/slider_config.yaml
ros2 run move_joint move_joint --ros-args -r /joint_setpoint:=/setpoint -p joint_name:=right_e0
```

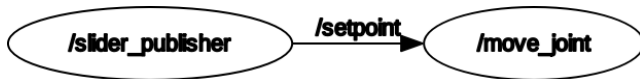
```
from simple_launch import SimpleLauncher

def generate_launch_description():

    sl = SimpleLauncher()
    sl.node('slider_publisher', 'slider_publisher',
            arguments = [sl.find('my_pkg', 'slider_config.yaml')])

    sl.node('move_joint', 'move_joint',
            remappings={'/joint_setpoint': '/setpoint'},
            parameters={'joint_name': 'right_e0'})

    return sl.launch_description()
```



Basic launch file that runs turtlesim with a control node

```
def generate_launch_description():

    ld = LaunchDescription()

    # run turtlesim (will spawn turtle1)
    sim_node = Node(package='turtlesim', executable='turtlesim_node')
    ld.add_action(sim_node)

    # declare a (Boolean) argument
    ld.add_action(DeclareLaunchArgument('manual', default_value=False))
    manual = LaunchConfiguration('manual')

    # open-loop node
    loop_node = Node(package='anf_launch', executable='loop', condition = UnlessCondition(manual))

    # manual node
    slider_config = f"{lookup('anf_launch')}/launch/Turtle.yaml"
    slider_node = Node(package='slider_publisher', executable='slider_publisher', name='turtle1',
                      condition = IfCondition(manual),
                      arguments = [slider_config])

    # namespaced group with those 2 nodes
    namespaced = GroupAction([PushRosNamespace('turtle1'), loop_node, slider_node])
    ld.add_action(namespaced)

    return ld
```

Basic launch file that runs turtlesim with a control node

```
from launch import LaunchDescription
from launch_ros.actions import Node, PushRosNamespace
from launch.actions import DeclareLaunchArgument, GroupAction
from launch.conditions import IfCondition, UnlessCondition
from launch.substitutions import LaunchConfiguration
from ament_index_python.packages import get_package_share_directory as lookup

def generate_launch_description():

    ld = LaunchDescription()

    # run turtlesim (will spawn turtle1)
    sim_node = Node(package='turtlesim', executable='turtlesim_node')
    ld.add_action(sim_node)

    # declare a (Boolean) argument
    ld.add_action(DeclareLaunchArgument('manual', default_value=False))
    manual = LaunchConfiguration('manual')

    # open-loop node
    loop_node = Node(package='anf_launch', executable='loop', condition = UnlessCondition(manual))

    # manual node
    slider_config = f"{lookup('anf_launch')}/launch/Turtle.yaml"
    slider_node = Node(package='slider_publisher', executable='slider_publisher', name='turtle1',
                      condition = IfCondition(manual),
                      arguments = [slider_config])

    # namespaced group with those 2 nodes
    namespaced = GroupAction([PushRosNamespace('turtle1'), loop_node, slider_node])
    ld.add_action(namespaced)

    return ld
```

Exposes a much lighter interface to write launch files

```
from simple_launch import SimpleLauncher

def generate_launch_description():

    sl = SimpleLauncher(use_sim_time = False)
    sl.declare_arg('manual', False)

    # run turtlesim with turtle1
    sl.node('turtlesim', 'turtlesim_node')

    # run the open-loop or manual control
    with sl.group(ns='turtle1'):

        with sl.group(unless_arg='manual'):
            # open loop control in this block
            sl.node('anf_launch', 'loop')

        with sl.group(if_arg='manual'):
            # manual control
            sl.node('slider_publisher', 'slider_publisher', name='turtle1',
                    arguments=[sl.find('anf_launch', 'Turtle.yaml')])

    return sl.launch_description()
```



https://github.com/oKermorgant/simple_launch

Using containers: official composition example

```
import launch
from launch_ros.actions import ComposableNodeContainer
from launch_ros.descriptions import ComposableNode

def generate_launch_description():
    container = ComposableNodeContainer(
        name='my_container',
        package='rclcpp_components',
        executable='component_container',
        composable_node_descriptions=[
            ComposableNode(
                package='composition',
                plugin='composition::Talker',
                name='talker'),
            ComposableNode(
                package='composition',
                plugin='composition::Listener',
                name='listener')
        ],
    )
    return launch.LaunchDescription([container])
```

Using containers: official composition example

```
import launch
from launch_ros.actions import ComposableNodeContainer
from launch_ros.descriptions import ComposableNode

def generate_launch_description():
    container = ComposableNodeContainer(
        name='my_container',
        package='rclcpp_components',
        executable='component_container',
        composable_node_descriptions=[
            ComposableNode(
                package='composition',
                plugin='composition::Talker',
                name='talker'),
            ComposableNode(
                package='composition',
                plugin='composition::Listener',
                name='listener')
        ],
    )
    return launch.LaunchDescription([container])
```

```
from simple_launch import SimpleLauncher

def generate_launch_description():
    sl = SimpleLauncher()

    # load Talker and Listener into new container
    with sl.container(name='my_container'):
        sl.node(package='composition', plugin='Talker', name='talker')
        sl.node(package='composition', plugin='Listener', name='listener')

    return sl.launch_description()
```

Passing parameters in Python launch files

```
# Node arguments are lists
Node(package='rviz2', executable='rviz2', name='rviz2',
      arguments=['-d', rviz_config_dir])

# Node parameters are a list of single-entry dictionaries
Node(package, executable, name,
      parameters=[{'use_sim_time': True}, {'robot_description': urdf_xml}])

# Node remappings are a list of (key, value) pairs
Node(package, executable, name,
      remappings=[('/cmd_vel', 'command'), ('/pose', 'odom')])

# Included launch parameters are a list of (key, value) pairs
IncludeLaunchDescription(
    AnyLaunchDescriptionSource('included_launch.py'),
    launch_arguments=[('namespace', namespace), ('use_sim_time', 'true')])
```

simple_launch: dictionaries everywhere except arguments

```
# Node arguments are still lists
sl.node(package='rviz2', executable='rviz2', name='rviz2',
        arguments=['-d', rviz_config_dir])

# Node parameters
sl.node(package, executable, name,
        parameters={'use_sim_time': True, 'robot_description': urdf_xml})

# Node remappings
sl.node(package, executable, name,
        remappings={'/cmd_vel': 'command', '/pose': 'odom'})

# Included launch parameters
sl.include(package, 'included_launch.py',
          launch_arguments={'namespace': namespace, 'use_sim_time': 'true'})
```

The main voodoo of Python launch files: Arguments are not classical Python types

```
sl.declare_arg('namespace', 'turtle')

# try to build topic '<namespace>/odom'
odom_topic = sl.arg('turtle') + '/odom' # does not work

odom_topic = sl.path_join(sl.arg('turtle'), 'odom') # ok
```

```
sl.declare_arg('urdf', 'robot.urdf')

urdf_path = '/path/to' + sl.arg('urdf') # nope

sl.node('robot_state_publisher',
        parameters = Command(['xacro', '/path/to', sl.arg('urdf')])) # ok

# does the same
sl.robot_state_publisher('pkg', 'sub-folder', sl.arg('urdf'))
```

Now for a few examples

