6.2 Composition of nodes

To try out compositions (we will not write code for composition since it is a more advanced topic[1][2]), we will use the 'rclcpp_components' package. Now we can check what kind of compositions we have already:

```
$ ros2 component types
```

```
(... components of other packages here)
composition
  composition::Talker
  composition::Listener
  composition::Server
  composition::Client
```

After this, we will start the container that contains the two (already familiar) nodes: "Talker" and "Listener".

```
$ ros2 run rclcpp_components component_container
```

In a second terminal we can check the start of the container:

```
$ ros2 component list
```

It has started:

```
/ComponentManager
```

```
/ComponentManager
```

Now we can start running our nodes from the second terminal:

```
$ ros2 component load /ComponentManager composition composition::Talker
```

```
$ ros2 component load /ComponentManager composition composition::
    Listener
```

We will get back a feedback like the following:

```
Loaded component 1 into '/ComponentManager' container node as '/talker'
Loaded component 2 into '/ComponentManager' container node as '/listener
    '
```

If we now list the component we can see that they have been added to the component manager:

```
$ ros2 component list
```

```
/ComponentManager
   1  /talker
   2  /listener
```

And also if we check the first tab where we started the container, we see the familiar counting and listening in the same tab. We can do these steps at once with a launch file[1] as well. (after clearing our terminals)

[1] A good explanation on YouTube from Robotics-Back End.
[2] The original codes can be found in the demo repository.
[1] Script of the launch file.

Work in progress, not for printing

```
$ ros2 launch composition composition_demo.launch.py
```

To unload the components we need to refer to their container and their IDs:

```
$ ros2 component unload /ComponentManager 1 2
```

```
Unloaded component 1 from '/ComponentManager' container
Unloaded component 2 from '/ComponentManager' container
```

Now You have a basic understanding in components, from which one can build its understanding to the more advanced nuances of node composition. We, however, move forward to the topic of the launch files.

6.3 Launch files

Now we will focus on how to make a launch file capable of handling basic tasks. We will use Python-based launch files. First, we will create a simple standalone launch file, based on the 'turtlesim' package. Let's make a dictionary for the launch files called "launch". Then inside this directory let's create a Python file called "turtlesim_mimic_launch.py". In this file, we will start three nodes simultaneously, two 'turtlesim_node' and a node called 'mimic', which will help us to make one turtle mimic the other. Now let's see the structure:

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

def generate_launch_description():
    return LaunchDescription([
    ])
```

We import the Python launch modules and define the launch descriptor generator, giving us back a list (of nodes to execute). Now let's fill up the list with the two 'turtlesim_node':

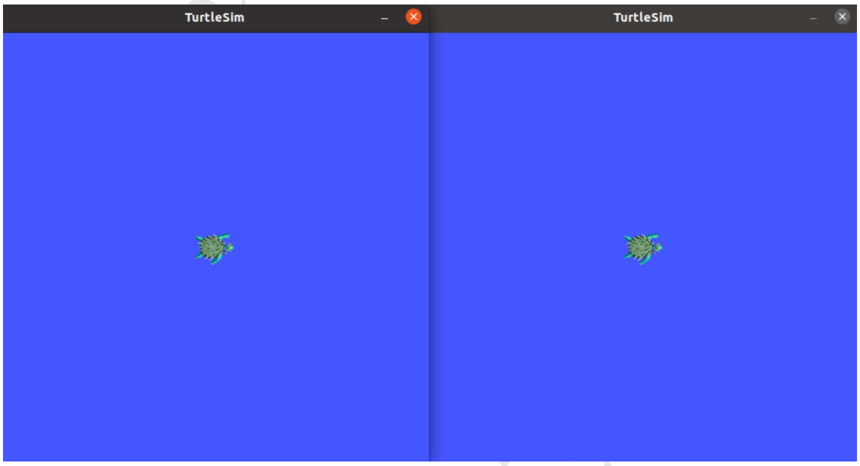
```
def generate_launch_description():
    return LaunchDescription([
        Node(
            package='turtlesim',
            namespace='turtlesim1',
            executable='turtlesim_node',
            name='sim'
        ),
        Node(
            package='turtlesim',
            namespace='turtlesim2',
            executable='turtlesim_node',
            name='sim'
        ),
        Node(
            package='turtlesim',
            executable='mimic',
            name='mimic',
            remappings=[
                ('/input/pose', '/turtlesim1/turtle1/pose'),
```

```
                    ('/output/cmd_vel', '/turtlesim2/turtle1/cmd_vel'),
               ]
          )
     ])
```

As it can be seen, we start two 'turtle_sim' nodes. The only difference is their namespaces, this way we can control them separately. The third node is a little more interesting. Now, 'mimic' can be thought of as an interface that helps make the remapping easier. With the 'remappings' we will remap (or "rename") it's '/input/pose' topic to which it is subscribed to the position of the first turtlesim's turtle '/turtlesim1/turtle1/pose'. After that we also remap the nem of the topic, for which it is publishing from the '/output/cmd_vel' to the second turtlesim's turtle: '/turtlesim2/turtle1/cmd_vel'.

If the file is completed, then we can move to the directory, and launch it:

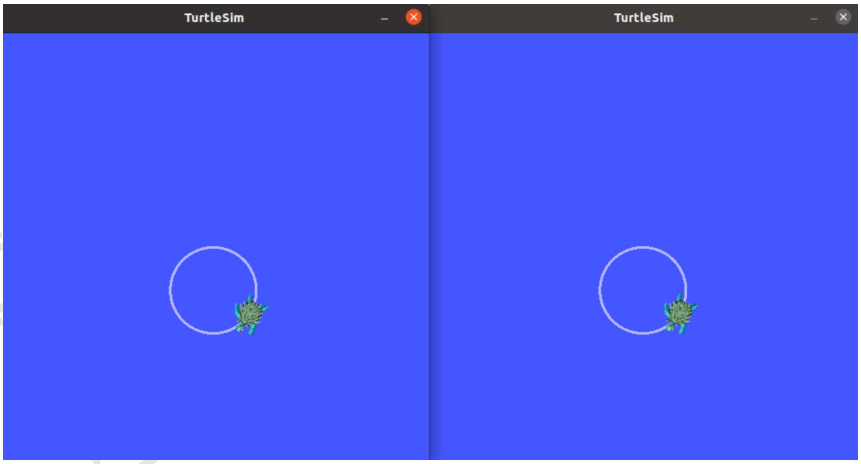
```
$ cd launch
$ ros2 launch turtlesim_mimic_launch.py
```

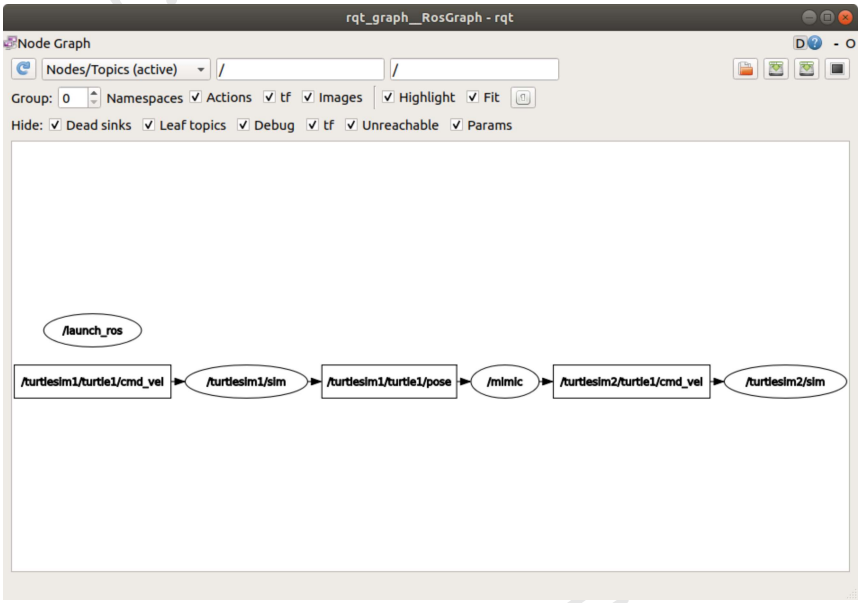


We can publish the first turtle's topic to move in a circle:

```
$ ros2 topic pub -r 1 /turtlesim1/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}}"
```

It is also visible if we run the 'rqt_graph'





Sometimes it is logical to include the launch file in a package that we are developing (for example if multiple nodes will be opened regularly when the package is deployed). Now we will see how to make this happen, navigate to the 'src' directory inside the workspace, and let's create a package:

```
$ ros2 pkg create launch_test --build-type ament_python
```

Now create a directory inside the package for the launch files called "launch" (this is the convention). We move on to the creation of the launch file inside this directory, and name it "script_launch.py":

```
import launch
import launch_ros.actions

def generate_launch_description():
    return launch.LaunchDescription([
        launch_ros.actions.Node(
            package='demo_nodes_cpp',
            executable='talker',
```

```
            name='talker'),
  ])
```

This is not a very complex script, it only starts the 'talker' node of the demo packages. To be able to use the launch files inside this directory, we need to update our 'setup.py' as follows:

```
import os                #ADDED
from glob import glob    #ADDED
from setuptools import setup

package_name = 'launch_test'

setup(
    name=package_name,
    version='0.0.0',
    packages=[package_name],
    data_files=[
        ('share/ament_index/resource_index/packages',
            ['resource/' + package_name]),
        ('share/' + package_name, ['package.xml']),
        (os.path.join('share', package_name, 'launch'), glob(os.path.
    join('launch', '*launch.[pxy][yma]*'))),   #ADDED
    ],
    install_requires=['setuptools'],
    zip_safe=True,
    maintainer='mark',
    maintainer_email='m4rk.domonkos@gmail.com',
    description='TODO: Package description',
    license='TODO: License declaration',
    tests_require=['pytest'],
    entry_points={
        'console_scripts': [
        ],
    },
)
```

Here we import the 'os' and the 'glob' for easy file management, then add the launch directory to our 'data_files' variables.

After building this package we are ready to test the functionality:

```
$ ros2 launch launch_test script_launch.py
```

This is the basics of adding a launch file to Your package. Now You can experiment with having other node definitions (scripts) inside this package for a more first-hand experience.

Occasionally, You will need to have more flexibility in Your launch files. Maybe You need different arguments to pass or parameters based on Your actual needs. This is when substitutions come in handy. To illustrate this, we will make two launch files. The first one will be able to dynamically include another launch file (substitutions.launch.py) and pass it some custom configuration values. Let's create a new launch file in our 'launch' directory called "parent.launch.py". The import part will look like this:

```
from launch_ros.substitutions import FindPackageShare

```

```
from launch import LaunchDescription
from launch.actions import IncludeLaunchDescription
from launch.launch_description_sources import
    PythonLaunchDescriptionSource
from launch.substitutions import PathJoinSubstitution, TextSubstitution
```

With the imports we acquire the following functionalities in order: 'FindPackageShare' will find us the share directory of a package; 'LaunchDescription' is the class of the launch description; 'IncludeLaunchDescription' is an action that will include another launch file to ours; 'PythonLaunchDesciptionSource' tells the system, that the included file is python based; and finally the two substitution we will use in this file: 'PathJoinSubstitusion' simply combines path elements, and 'TextSubstitution' will substitute a text value.

Now, inside the 'generate_launch_description()' we define a dictionary with one element, that we will later use. We will return inside our LaunchDescription two things, an inclusion of the other launch file, and an argument list.

```
def generate_launch_description():
    colors = {
        'background_r': '200'
    }

    return LaunchDescription([
        IncludeLaunchDescription(
            PythonLaunchDescriptionSource([
                PathJoinSubstitution([
                    FindPackageShare('launch_test'),
                    'launch',
                    'substitutions1.launch.py'
                ])
            ]),
            launch_arguments={
                'turtlesim_ns': 'turtlesim2',
                'use_provided_red': 'True',
                'new_background_r': TextSubstitution(text=str(colors['
    background_r']))
            }.items()
        )
    ])
```

For the inclusion (reading outwards), we will use the 'FindPackageShare' to find the "launch_test' package's share directory, which contains all the non-source files of the package. When we have this, we will use the 'PathJoinSubstitution' to join this path with the launch directory and the filename of the launch file that we want to include. Then with 'PythonLaunchDescriptionSource' we tell the system that this file contains a Pyhton-based launch description, and finally, we include it.

In the 'launch_arguments' dictionary are the parameters that we intend to pass to the other launch file. We will pass the name of the turtlesim's namespace, a truth value that will describe whether the launch should use the original red component of the background's color channel, and a substitution of the background color (in the case of the truth value will be false).

Now we create the launch file that will launch the turtlesims, which will be given new

parameters.

```
from launch_ros.actions import Node

from launch import LaunchDescription
from launch.actions import DeclareLaunchArgument, ExecuteProcess,
    TimerAction
from launch.conditions import IfCondition
from launch.substitutions import LaunchConfiguration, PythonExpression


def generate_launch_description():

    #TBF ...

    return LaunchDescription([
        turtlesim_ns_launch_arg,
        use_provided_red_launch_arg,
        new_background_r_launch_arg,
        turtlesim_node,
        spawn_turtle,
        change_background_r,
        TimerAction(
            period=2.0,
            actions=[change_background_r_conditioned],
        )
    ])
```

Among the used imports, we import the 'IfCondition' to be able to use conditionals inside the launch file, and 'LaunchConfiguration' and 'PythonExpression'. In the generator function, we will fill up the missing variable definitions, but for easier understanding let's just observe what we are returning in the descriptor. We will make use of the three parameters: namespace of the turtlesim, the boolean variable of the provision of the red color channel value, and the value of the red color. We will start a turtlesim node, spawn a new turtle to it (now we will have two), and we will change the background's color. Finally, if a timer is done counting (after 2 seconds) we conditionally change the background's color again. Now let's see the definitions of these.

```
turtlesim_ns = LaunchConfiguration('turtlesim_ns')
use_provided_red = LaunchConfiguration('use_provided_red')
new_background_r = LaunchConfiguration('new_background_r')

turtlesim_ns_launch_arg = DeclareLaunchArgument(
    'turtlesim_ns',
    default_value='turtlesim1'
)
use_provided_red_launch_arg = DeclareLaunchArgument(
    'use_provided_red',
    default_value='False'
)
new_background_r_launch_arg = DeclareLaunchArgument(
    'new_background_r',
    default_value='200'
)
```

BETA

The first three lines take the value of the launch argument in any part of the launch description. Then we define the launch arguments to pass from above or from the console.

Then we start the 'turtlesim_node', with the defined namespace.

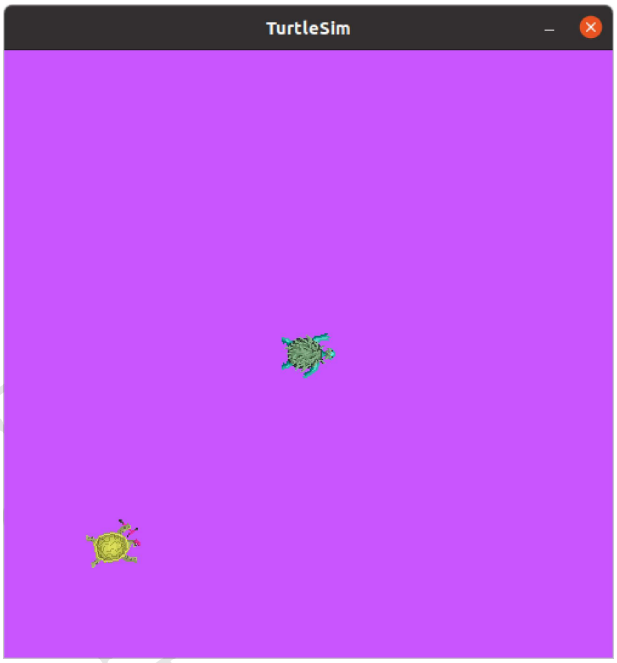
```
turtlesim_node = Node(
    package='turtlesim',
    namespace=turtlesim_ns,
    executable='turtlesim_node',
    name='sim'
)
```

Then we execute a service call (to spawn the second turtle with the same namespace and to a predefined coordinateset) with the 'ExecuteProcess'

```
spawn_turtle = ExecuteProcess(
    cmd=[[
        'ros2 service call ',
        turtlesim_ns,
        '/spawn ',
        'turtlesim/srv/Spawn ',
        '"{x: 2, y: 2, theta: 0.2}"'
    ]],
    shell=True
)
```

With similar logic, we will call a service to change the background to a predefined value (with the red channel set to 120), and in our final definition we change the color of the background again, but only if the the parameters equal to 200 and "True". We make this conditional definition in a pythonic way, hence the used substitution.

```
change_background_r = ExecuteProcess(
    cmd=[[
        'ros2 param set ',
        turtlesim_ns,
        '/sim background_r ',
        '120'
    ]],
    shell=True
)
change_background_r_conditioned = ExecuteProcess(
    condition=IfCondition(
        PythonExpression([
            new_background_r,
            ' == 200',
            ' and ',
            use_provided_red
        ])
    ),
    cmd=[[
        'ros2 param set ',
        turtlesim_ns,
        '/sim background_r ',
        new_background_r
    ]],
    shell=True
```




```
)
```

For the full version of the files see A.3.1 A.3.2, and for even more detail on substitution, You can read the documentation[1].

Now we can build the package and try out our launchfiles. First, we launch our parent file without any extra provided arguments.

```
$ ros2 launch launch_test parent.launch.py
```

This will execute the 'substitution.launch.py' with the provided parameter values (namespace = "turtlesim2", conditional = True, background's red = 200). We will see two turtles and three stages of background (blue, pink, and lilac). The final stage should look like this:

Now, we can try this out with other arguments. If we are not sure about the arguments we can ask for some help:

```
$ ros2 launch launch_test substitutions1.launch.py --show-args
```

```
Arguments (pass arguments as '<name>:=<value>'):

    'turtlesim_ns':
        no description given
        (default: 'turtlesim1')

    'use_provided_red':
        no description given
        (default: 'False')

    'new_background_r':
        no description given
        (default: '200')
```

[1]ROS2 Launch documentation

First, let's try with the "correct" values for the conditional (after ending the previous execution):

```
$ ros2 launch launch_test substitutions1.launch.py turtlesim_ns:='
    turtlesim3' use_provided_red:='True' new_background_r:=200
```

Now let's try something different (after ending the previous execution), we start our (not parent!) launch file with some argument values:

```
$ ros2 launch launch_test substitutions1.launch.py turtlesim_ns:='
    turtlesim3' use_provided_red:='False' new_background_r:=150
```

And this is the result (only one change):

The last topic regarding launch files is the topic of event handlers. This concept can come in handy when we want our system to behave in different ways whenever an already registered event happens. To demonstrate this, we will change our script inside 'substitutions1.launch.py', for which we make a new file called "eventhandler.launch.py" and copy the content of the prior first. Now the script will be the same for most cases except for two parts: the imports and the return. Below is visible only the changed part, for the full version see A.3.3

```
from launch_ros.actions import Node

from launch import LaunchDescription
from launch.actions import (DeclareLaunchArgument, EmitEvent,
    ExecuteProcess,
                            LogInfo, RegisterEventHandler, TimerAction)
from launch.conditions import IfCondition
from launch.event_handlers import (OnExecutionComplete, OnProcessExit,
                                   OnProcessIO, OnProcessStart, OnShutdown)
from launch.events import Shutdown
from launch.substitutions import (EnvironmentVariable, FindExecutable,
```

```
                                LaunchConfiguration, LocalSubstitution,
                                PythonExpression)


def generate_launch_description():
    #SIMILAR TO THE PREVIOUS
    # ...

    return LaunchDescription([
        turtlesim_ns_launch_arg,
        use_provided_red_launch_arg,
        new_background_r_launch_arg,
        turtlesim_node,
        RegisterEventHandler(
            OnProcessStart(
                target_action=turtlesim_node,
                on_start=[
                    LogInfo(msg='Turtlesim started, spawning turtle'),
                    spawn_turtle
                ]
            )
        ),
        RegisterEventHandler(
            OnProcessIO(
                target_action=spawn_turtle,
                on_stdout=lambda event: LogInfo(
                    msg='Spawn request says "{}"'.format(
                        event.text.decode().strip())
                )
            )
        ),
        RegisterEventHandler(
            OnExecutionComplete(
                target_action=spawn_turtle,
                on_completion=[
                    LogInfo(msg='Spawn finished'),
                    change_background_r,
                    TimerAction(
                        period=2.0,
                        actions=[change_background_r_conditioned],
                    )
                ]
            )
        ),
        RegisterEventHandler(
            OnProcessExit(
                target_action=turtlesim_node,
                on_exit=[
                    LogInfo(msg=(EnvironmentVariable(name='USER'),
                            ' closed the turtlesim window')),
                    EmitEvent(event=Shutdown(
                        reason='Window closed'))
                ]
            )
```

```
        ),
        RegisterEventHandler(
            OnShutdown(
                on_shutdown=[LogInfo(
                    msg=['Launch was asked to shutdown: ',
                         LocalSubstitution('event.reason')]
                )]
            )
        ),
    ])
```

We import an event called 'shutdown' and some commonly used event handlers. The return was the same until we started the turtlesim node. Now we will use the event handler to print logs to the terminal. For this

The first registration's function is simply when the the turtlesim node starts then it send out a log message, and then starts the spawning to the second trutle.

```
RegisterEventHandler(
    OnProcessStart(
        target_action=turtlesim_node,
        on_start=[
            LogInfo(msg='Turtlesim started, spawning turtle'),
            spawn_turtle
        ]
    )
),
```

The second registration is used as a callback function. When the spawn of the second turtle happens it sends an IO message to the terminal. When this event occurs, it is watched, and we can attach additional behavior (currently we change the message) to it.

```
RegisterEventHandler(
    OnProcessIO(
        target_action=spawn_turtle,
        on_stdout=lambda event: LogInfo(
            msg='Spawn request says "{}"'.format(
                event.text.decode().strip())
        )
    )
),
```

We can register an event handler when a process is completed (successfully), here if this happens, we print it out as a log and change the background (once or twice based on the values of the arguments, similarly like previously we did when leading about the substitutions).

```
RegisterEventHandler(
    OnExecutionComplete(
        target_action=spawn_turtle,
        on_completion=[
            LogInfo(msg='Spawn finished'),
            change_background_r,
            TimerAction(
                period=2.0,
                actions=[change_background_r_conditioned],
```

```
            )
        ]
    )
),
```

Then if the process (turtlesim) exits we can initiate a shutdown inside our event handler:

```
RegisterEventHandler(
    OnProcessExit(
        target_action=turtlesim_node,
        on_exit=[
            LogInfo(msg=(EnvironmentVariable(name='USER'),
                    ' closed the turtlesim window')),
            EmitEvent(event=Shutdown(
                reason='Window closed'))
        ]
    )
),
```

And finally, if the shutdown happens, we log a message about the shutdown.

```
RegisterEventHandler(
    OnShutdown(
        on_shutdown=[LogInfo(
            msg=['Launch was asked to shutdown: ',
                LocalSubstitution('event.reason')]
        )]
    )
),
```

After building our project, we can run the launch file:

```
$ ros2 launch launch_test eventhandler.launch.py turtlesim_ns:='
    turtlesim3' use_provided_red:='True' new_background_r:=200$
```

This is what concludes our current topic of launch files. Now You should be able to have a basic understanding of reading and writing launch files. It can be seen that launch files come in handy in various ways, especially when managing larger project.[2]

[2]Some advice on managing larger projects

Unofficial pdf of EE Dept.AI