UNIVERSITÀ
DI TRENTO
Department of
Information Engineering and Computer Science

ROS 2 Development

Edoardo Lamon, Luigi Palopoli, Enrico Saccon

Software Development for Collaborative Robots

Academic Year 2025/26

ROS 2 Nodes

Launch files

Launch Files

- ROS 2 Launch files allow you to start up and configure a number of executables containing ROS 2 nodes simultaneously.

- Launch files written in *Python*, *XML*, or *YAML* can start and stop different nodes as well as trigger and act on various events.

- As standard, launch files are located in the *launch* folder inside the package.

Write a Launch File

PYTHON

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

def generate_launch_description():
  return LaunchDescription([
    Node(
      package="chatters",
      executable="mytalker",
      name="mytalker_launch",
      parameters=[
        {"rate"  : 5000}
      ],
      output="screen",
      emulate_tty=True
      remappings=[
       ('/topic', '/topic_launch'),
    )
  ])
```

Write a Launch File

PYTHON

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

def generate_launch_description():
  return LaunchDescription([
      Node(
          package="chatters",
          executable="mytalker",
          name="mytalker_launch",
          parameters=[
              {"rate"  : 5000}
          ],
          output="screen",
          emulate_tty=True
          remappings=[
           ('/topic', '/topic_launch'),
      )
  ])
```

YAML

```yaml
launch:
 - node:
   pkg: "chatters"
   exec: "mytalker"
   name: "mytalker_yaml"
   output: "screen"
   remap:
    -
      from: "/my_topic"
      to: "/default_topic"
param:
  -
    name: "rate"
    value: 5000
```

Write a Launch File

PYTHON

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

def generate_launch_description():
  return LaunchDescription([
    Node(
      package="chatters",
      executable="mytalker",
      name="mytalker_launch",
      parameters=[
        {"rate"  : 5000}
      ],
      output="screen",
      emulate_tty=True
      remappings=[
       ('/topic', '/topic_launch'),
    )
  ])
```

YAML

```yaml
launch:
 - node:
   pkg: "chatters"
   exec: "mytalker"
   name: "mytalker_yaml"
   output: "screen"
   remap:
    -
       from: "/my_topic"
       to: "/default_topic"
param:
   -
     name: "rate"
     value: 5000
```

XML (ROS 1-like)

```xml
<launch>

 <arg name=“rate" default=“5000"/>

 <node pkg="chatters" exec="mytalker" name="mytalker_xml" output='screen’>
   <param name="rate " value="$(var rate)"/>
   <remap from="/topic" to="$(var topic)"/>
 </node>

</launch>
```

Python, XML, or YAML: Which should I use?

- Launch files in ROS 1 were written in XML, so XML may be the most familiar to people coming from ROS 1.

- For most applications, the choice of which ROS 2 launch format comes down to developer preference.

- Using Python for ROS 2 launch is more flexible because of following two reasons:
  - Python is a scripting language, and thus you can leverage the language and its libraries in your launch files;
  - ros2/launch (general launch features) and ros2/launch_ros (ROS 2 specific launch features) are written in Python and thus you have lower level access to launch features that may not be exposed by XML and YAML.

Running the Launch File

- Directly in the *launch* folder (works without installation):

```
cd launch
ros2 launch <launch_file>
```

- Provided by a package (requires installation - RECCOMENDED):

```
ros2 launch <package_name> <launch_file>
```

  In the CMakeLists.txt add:

```
install(
     DIRECTORY launch
     DESTINATION share/${PROJECT_NAME}
   )
```

- In ROS 2 the launch file goes with his extension (.py/.xml/.yaml). To ensure ensures that all launch file formats are recognized, add in package.xml:

```
<exec_depend>ros2launch</exec_depend>
```

Exercise

Add the subscriber to the nodes to be launched in each launch file (Python, YAML, XML).

Launch Different Nodes - Python

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

def generate_launch_description():
    talker_node = Node(
        package="chatters",
        executable="mytalker",
        name="mytalker_launch",
        output="screen",
        emulate_tty=True,
        parameters=[
            {"topic" : "topic_launch"},
            {"rate"  : 500}
        ]
    )
    listener_node = Node(
        package="chatters",
        executable="mylistener",
        name="mylistener_launch",
        output="screen",
        emulate_tty=True,
        parameters=[
            {"topic" : "topic_launch"}
        ]
    )

    return LaunchDescription([
        talker_node,
        listener_node
    ])
```

Launch Different Nodes - YAML

```
launch:
 - node:
   pkg: "chatters"
   exec: "mytalker"
   name: "mytalker_yaml"
   output: "screen"
   param:
   -
     name: "topic"
     value: "topic_yaml"
   -
     name: "rate"
     value: 5000

 - node:
   pkg: "chatters"
   exec: "mylistener"
   name: "mylistener_yaml"
   output: "screen"
   param:
   -
     name: "topic"
     value: "topic_yaml"
```

Launch Different Nodes - XML

```
<launch>
 <arg name="topic_name" default="topic_launch_xml"/>

 <node pkg="custom_pkg" exec="custom_pub_standalone" name="my_publisher_launch_xml"
output='screen'>
   <param name="topic_name" value="$(var topic_name)"/>
 </node>

 <node pkg="custom_pkg" exec="custom_sub_standalone" name="my_subscriber_launch_xml"
output='screen
     <remap from="/my_topic" to="$(var topic_name)"/>
 </node>


</launch>
```

Setting Arguments

From command line it is possible to pass arguments to the launch file with key:=value syntax:

```
ros2 launch <package_name> <launch_file_name> key:=value
```

You need to explicitly declare them. In Python:

```
from launch.actions import DeclareLaunchArgument
from launch.substitutions import LaunchConfiguration, TextSubstitution
def generate_launch_description():
    topic_name_arg = DeclareLaunchArgument("topic_name_arg", default_value=TextSubstitution(text="topic_launch"))
    pub_node = Node(
        package='custom_pkg',
        executable='custom_pub_standalone',
        name='my_publisher_launch',
                          parameters=[{'topic_name' : LaunchConfiguration('topic_name_arg’)}] )
    return LaunchDescription([
        topic_name_arg,
        pub_node,])
```

Load Parameters File

- We know it is possible to pass to a node a YAML file which contains parameters:

```
ros2 run <pkg> <node> --ros-args --params-file <path to YAML file>
```

- Can we pass it also to a launch file?

Load Parameters File

Can we pass it also to a launch file? YES

```
import os
from ament_index_python import get_package_share_directory

parameters_file = os.path.join(get_package_share_directory('custom_pkg'),
'config','my_publisher.yaml’)

pub_node = Node(
    package='custom_pkg',
    executable='custom_pub_standalone',
    name='my_publisher_launch',
    parameters=[parameters_file]
)
```

Load Parameters File

Can we pass it also to a launch file? YES

CAVEATS:

- Make sure the YAML file is installed, otherwise the it won’t be found. In the CMakeLists.txt:

  ```
  install(
  DIRECTORY config
  DESTINATION share/${PROJECT_NAME})
  ```

- Make sure the name of the node (e.g. 'my_publisher_launch'), matches the one in the YAML file.

Include Another Launch

It is also possible to nest the call of another launch file:

```
from launch.launch_description_sources import PythonLaunchDescriptionSource, AnyLaunchDescriptionSource

# include another launch file
    launch_include_py = IncludeLaunchDescription(
        PythonLaunchDescriptionSource(os.path.join(
            get_package_share_directory('custom_pkg'),'launch','publisher_launch_python.py'))
    )
    launch_include_yaml = IncludeLaunchDescription(
        AnyLaunchDescriptionSource(os.path.join(
            get_package_share_directory('custom_pkg'),'launch','publisher_launch_yaml.yaml'))
    )
    return LaunchDescription([
        launch_include_py,
        launch_include_yaml
    ])
```

ROS 2 Nodes

Services: clients and servers

Services

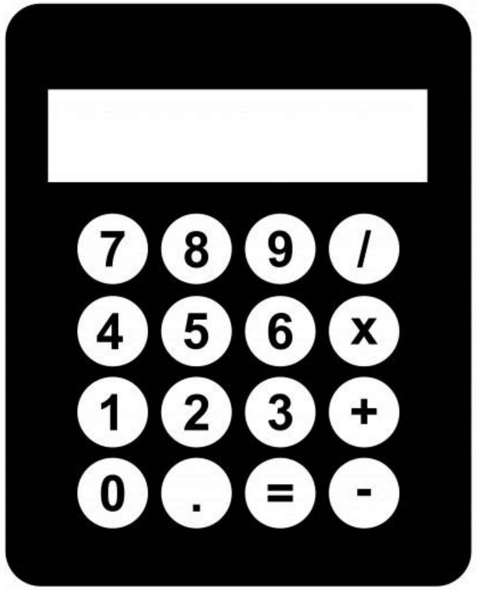
- Actions to be carried out in a **small amount** of time
- In ROS1 services were **synchronous** (client blocking call)
  - If a server did not return a response, the caller would stay hanging ;
  - The server must return an answer almost immediately.
- In ROS2 services are **asynchronous**
  - The response is returned through a future object;
  - The client can continue its execution and check for the result whenever it prefers.
- Best practice: in the client, wait for the response but **set a timeout**

Services

- **Use** if:
  - The data that the server sends to the client *depend* on the request
  - The client needs to receive confirmation that the server processed the data correctly, otherwise an error is returned
  - The client needs to verify that the server is ready to process the message before sending it
- **Not use** if:
  - The server needs a lot of time to process the data → depends on your system
  - The client needs to receive information from the server during its execution

Writing the Service Server

- Let's build a node that acts as a calculator!
- We want it to be able to do:
  - Additions
  - Subtractions
  - Multiplications
  - Divisions!



Writing the Service Server

```
#include "rclcpp/rclcpp.hpp"
#include <memory>

class Calculator : public rclcpp::Node {
public:
 Calculator() : Node("calculator") {
   this->service_ = this->create_service<>(
     "calculator", std::bind(&Calculator::calc, this,
     std::placeholders::_1, std::placeholders::_2))
 }
private:
 void calc(const std::shared_ptr<::Request> request,
           std::shared_ptr<::Response> response){}
 rclcpp::Service<>::SharedPtr service_;
}
```

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

Writing the Service Server

```
#include "rclcpp/rclcpp.hpp"
#include <memory>

class Calculator : public rclcpp::Node {
public:
 Calculator() : Node("calculator") {
   this->service_ = this->create_service<>(
     "calculator", std::bind(&Calculator::calc, this,
     std::placeholders::_1, std::placeholders::_2))
 }
private:
 void calc(const std::shared_ptr<::Request> request,
           std::shared_ptr<::Response> response){}
 rclcpp::Service<>::SharedPtr service_;
}
```

What should we put here?

Writing the Service Server

```
#include "rclcpp/rclcpp.hpp"
#include <memory>

class Calculator : public rclcpp::Node {
public:
 Calculator() : Node("calculator") {
   this->service_ = this->create_service<>(
     "calculator", std::bind(&Calculator::calc, this,
     std::placeholders::_1, std::placeholders::_2))
 }
private:
 void calc(const std::shared_ptr<::Request> request,
           std::shared_ptr<::Response> response){}
 rclcpp::Service<>::SharedPtr service_;
}
```

```
string type
float64 a
float64 b
---
float64 res
```

Service Interface

- Where should it be placed? <package_name>/srv/Calc.srv
  - <package_name> → other package → BEST PRACTICE
  - <package_name> → same package

- How should it be added for compilation?

  Package.xml

  ```
  <buildtool_depend>rosidl_default_generators</buildtool_depend>
  <exec_depend>rosidl_default_runtime</exec_depend>
  <member_of_group>rosidl_interface_packages</member_of_group>
  ```

  CMakeLists.txt

  ```
  find_package(rosidl_default_generators REQUIRED)
  rosidl_generate_interfaces(${PROJECT_NAME} "srv/Calc.srv")
  ```

- Compile with colcon build and check that it is correct:

  ```
  $ source install/setup.bash && ros2 interface show <pkg_name>/srv/Calc
  ```

package/srv/Calc.srv

```
string type
float64 a
float64 b
---
float64 res
```

Service Interface

- Create a new package and a folder srv inside
- How should it be included in the source files?
- In CMakeLists.txt we need to add it as a dependency:

```
find_package(calculator_interfaces REQUIRED)
ament_target_dependencies(calculator_node rclcpp calculator_interfaces)
```

- If the package **is not** the same, then also package.xml should be informed

```
<depend>calculator_interfaces</depend>
```

- Include the headers which have the form

```
#include "<package_name>/srv/<interface_file_name>.hpp"
#include "calculator_interfaces/srv/calc.hpp"
```

Write the Service Server

```
#include "rclcpp/rclcpp.hpp"
#include <memory>
#include "calculator_interfaces/srv/calc.hpp"

class Calculator : public rclcpp::Node {
public:
 Calculator() : Node("calculator") {
   this->service_ = this->create_service<calculator_interfaces::srv::Calc>(
    "calculator", std::bind(&Calculator::calc, this,
    std::placeholders::_1, std::placeholders::_2));
 }
private:
 void calc(const std::shared_ptr<calculator_interfaces::srv::Calc::Request> request,
          std::shared_ptr<calculator_interfaces::srv::Calc::Response> response){}
 rclcpp::Service<calculator_interfaces::srv::Calc>::SharedPtr service_;
};
```

Write the Service Server

```
void calc(const std::shared_ptr<calculator_interfaces::srv::Calc::Request> request,
        std::shared_ptr<calculator_interfaces::srv::Calc::Response> response)
 {
  if (request->type == "sum"){
    response->res = request->a + request->b;
  }
  else if (request->type == "sub"){
    response->res = request->a - request->b;
  }
  else if (request->type == "mul"){
    response->res = request->a * request->b;
  }
  else if (request->type == "div"){
    response->res = request->a / request->b;
  }
  else {
    RCLCPP_ERROR(this->get_logger(), "Invalid operation type");
    return;
  }
  RCLCPP_INFO(this->get_logger(), "Operation: %s %s %s = %s",
    std::to_string(request->a).c_str(), request->type.c_str(),
    std::to_string(request->b).c_str(), std::to_string(response->res).c_str());
 }
```

Write the Service Client

```
#include "rclcpp/rclcpp.hpp"
#include "calculator_interfaces/srv/calc_msg.hpp“

using namespace std::chrono_literals;
using namespace calculator_interfaces::srv;
using namespace std::chrono;

const auto TIMEOUT = 1s;

int main(int argc, char **argv)
{
 rclcpp::init(argc, argv);

 if (argc != 4) {
    RCLCPP_INFO(rclcpp::get_logger("calculator_client"), "usage: calculator_client X T Y");
    return 1;
 }

 std::shared_ptr<rclcpp::Node> node = rclcpp::Node::make_shared("calculator_client");
 rclcpp::Client<Calc>::SharedPtr client =
   node->create_client<Calc>("calculator");
```

```
auto request = std::make_shared<Calc::Request>();
request->a = atoll(argv[1]);
request->b = atoll(argv[3]);

switch (argv[2][0]){
case '+':
  request->type = "sum"; break;
case '-':
  request->type = "sub"; break;
case '*':
  request->type = "mul"; break;
case '/':
  request->type = "div"; break;
default:
  RCLCPP_ERROR(rclcpp::get_logger("calculator_client"), "Invalid operation type");
  break;
}
while (!client->wait_for_service(TIMEOUT)) {
  if (!rclcpp::ok()) {
    RCLCPP_ERROR(rclcpp::get_logger("calculator_client"), "Interrupted while waiting for the service. Exiting.");
    return 0; }
  RCLCPP_INFO(rclcpp::get_logger("calculator_client"), "service not available, waiting again...");
}
```

Write the Service Client

```
auto result = client->async_send_request(request);

if (rclcpp::spin_until_future_complete(node, result) ==
  rclcpp::FutureReturnCode::SUCCESS)
{
  RCLCPP_INFO(rclcpp::get_logger("calculator_client"), "Result of %lf %s %lf = %lf",
    request->a, request->type.c_str(), request->b, result.get()->res);
} else {
  RCLCPP_ERROR(rclcpp::get_logger("calculator_client"), "Failed to call service  calculator");}
 rclcpp::shutdown();
return 0;
}
```

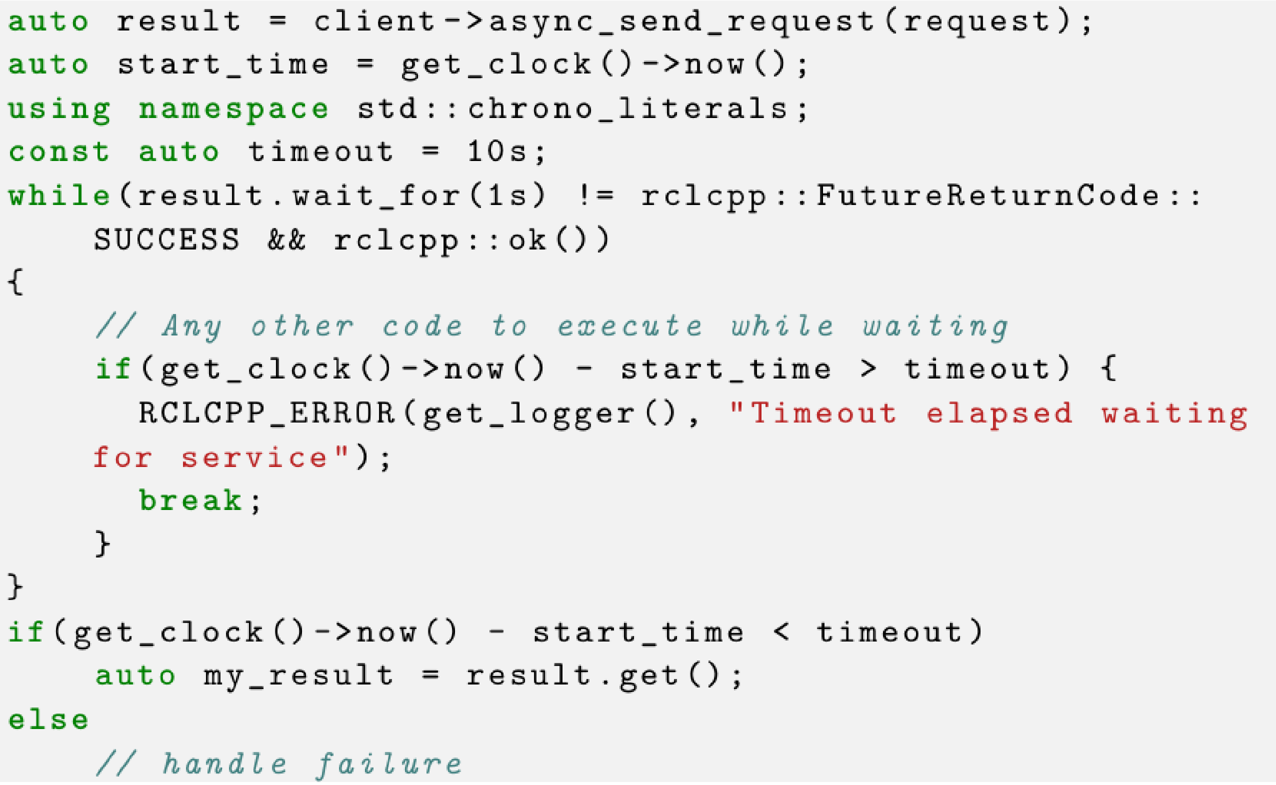
- Use always the timeout for receiving the response (in some examples it is not present)!

- To invoke the callbacks of subscriptions, timers, service servers, action servers, etc. on incoming messages and events on one or multiple threads, use an **Executor.**

Write the Service Client

- Be aware : spin_until_future_complete() can not be used if the node is added as a component to an **Executor;**

- In that case, one might store the result and check it from a timer with result.wait_for(DELAY) (non-blocking):



```
auto result = client->async_send_request(request);
auto start_time = get_clock()->now();
using namespace std::chrono_literals;
const auto timeout = 10s;
while(result.wait_for(1s) != rclcpp::FutureReturnCode::
    SUCCESS && rclcpp::ok())
{
    // Any other code to execute while waiting
    if(get_clock()->now() - start_time > timeout) {
      RCLCPP_ERROR(get_logger(), "Timeout elapsed waiting
    for service");
      break;
    }
}
if(get_clock()->now() - start_time < timeout)
    auto my_result = result.get();
else
    // handle failure
```

Exercise

- Create a launch file where both nodes are launched and the parameters are set as arguments in the launch file.