3. Pár egyszerű példa ROS2 node-ok C++ implementációjára

Kiegészítő anyag

Robot operációs rendszerek és fejlesztői ökoszisztémák

BMEVIIIAV55

Összeállította: Gincsainé Szádeczky-Kardoss Emese

szadeczky.emese@vik.bme.hu

Budapesti Műszaki és Gazdaságtudományi Egyetem

Irányítástechnika és Informatika Tanszék

2025

Bevezetés

Ebben a kiegészítő anyagban pár egyszerű példán keresztül mutatjuk be, hogy miként lehet ROS2 node-okat implementálni C++ nyelven. Itt nem szerepelnek olyan, az előadáson elhangzott ismeretek, hogy mik azok a node-ok, hogyan tudnak kommunikálni, csak az implementáció kérdéseit tárgyaljuk. Nem célunk minden lehetséges megoldás ismertetése, számos további (akár jobb, szebb) példa található sok helyen.

Feltételezzük, hogy az előző kiegészítő anyag alapján már létrehoztunk egy saját package-et (`my_pkg`) a colcon munkaterünkben (`colcon_ws`). Ebben fogunk most dolgozni. (Vagy minden új terminál megnyitásakor be kell állítani, hogy ez az aktuális munkakörnyezet (`source install/setup.bash`), vagy a `.bashrc` végére kell tenni a megfelelő sort.)

Első ROS node-unk

Kicsit módosítottunk a szokásos Hello World! mintapéldán. Nem ezt a szöveget, és nem direkt a terminálban fogjuk megjeleníteni. Egy publisher-t hozunk létre, ami egy topic-ra adott időközönként elküld egy statikus szöveget. A ROS1-es mintához hasonlóan most is a `my_pkg/src` könyvtárba hozzuk létre a **`my_first_publisher.cpp`** fájlt a következő kóddal:

```
#include "rclcpp/rclcpp.hpp"
#include "std_msgs/msg/string.hpp"

int main(int argc, char** argv)
{
    // Init ROS2
    rclcpp::init(argc, argv);
    // Create a node
    auto n = std::make_shared<rclcpp::Node>("my_first_node");
    // Display info for user
    RCLCPP_INFO(n->get_logger(), "First node is running.");
    // Set publishing rate in Hz
    rclcpp::Rate r(0.5);
    // Topic for a string
    rclcpp::Publisher<std_msgs::msg::String>::SharedPtr first_pub = n->
              create_publisher<std_msgs::msg::String>("my_topic", 1000);
    // Loop for sending
    while(rclcpp::ok())
    {
        // Message for the string
        auto msg1 = std_msgs::msg::String();
        // String to send
        msg1.data = "Hello KUKA, hello BME!";
        // Publish string message on the topic
        first_pub->publish(msg1);
        // Ensure desired rate
        r.sleep();
    }
    // Stop
    RCLCPP_INFO(n->get_logger(), "Stopping node.");
    rclcpp::shutdown();
    return 0;
}
```

Mielőtt build-elnénk a package-et, szükséges pár információt megadni a `my_pkg/CMakeLists.txt` állományban. Ehhez nyissuk meg, és keressük mega a `# find dependencies` részt (nagyjából a 8. sor). Itt megtaláljuk azokat a függőségeket, amiket a package létrehozásakor megadtunk. Ha valami akkor kimaradt, itt kell egy újabb sort beírni (pl. a `std_msgs`-t).

```
# find dependencies
find_package(ament_cmake REQUIRED)
find_package(rclcpp REQUIRED)
find_package(std_msgs REQUIRED)
```

Ezt követően adjuk hozzá az új node-unkhoz tartozó fájlt és függőségeit:

```
add_executable(my_first_node src/my_first_publisher.cpp)
ament_target_dependencies(my_first_node rclcpp std_msgs)
```

Továbbá szúrjuk be az alábbit:

```
install(TARGETS
   my_first_node
   DESTINATION lib/${PROJECT_NAME})
```

Az új függőség miatt a `package.xml`-t is ki kell egészíteni (kb. 12-13. sorok):

```
<depend>rclcpp</depend>
<depend>std_msgs</depend>
```

Most már build-elhetjük a package-et. Egy terminálban, a `colcon_ws` munkatér könyvtárában adjuk ki a `colcon build` utasítást:

```
ros_user@ubuntu:~/colcon_ws/src$ cd ..
ros_user@ubuntu:~/colcon_ws$ colcon build
```

Ha valamit elrontottunk, a kapott hibaüzenet alapján javítsuk! Amennyiben nincs hiba, futtassunk. Egy terminálban indítsuk el az új node-ot:

```
ros_user@ubuntu:~$ ros2 run my_pkg my_first_node
```

Ha jól dolgoztunk, akkor láthatjuk, hogy fut a node. Ha azt is ellenőrizni szeretnénk, hogy mi van a topic-on, akkor indítsunk egy másik terminált, és nézzük meg, hogy létezik-e a topic, és ha igen, akkor mi van rajta:

```
ros_user@ubuntu:~$ ros2 topic list
ros_user@ubuntu:~$ ros2 topic echo /my_topic
```

Ha minden jól működik, leállíthatjuk a node-ot (`Ctrl + C`), és már nem kell a topic-ot se hallgatni.

Első ROS node-unk kicsit másként

Írjuk át a fenti kódot úgy, hogy egy új osztályt hozunk létre a publikáló node-unk számára.

```
#include <chrono>
#include "rclcpp/rclcpp.hpp"
#include "std_msgs/msg/string.hpp"

class MyPublisher : public rclcpp::Node
{
    public:
    MyPublisher() : Node("my_first_node") // Constructor of the node
    {
        // Display info for user
```

```
        RCLCPP_INFO(this->get_logger(), "First node using class is
                                                                running.");
        // Topic for a string
        first_pub_ = this->create_publisher<std_msgs::msg::String>
                                                    ("my_topic", 1000);
        // Timer for sending
        timer_ = this->create_wall_timer(std::chrono::milliseconds(2000),
                            std::bind(&MyPublisher::timerCallback, this));
    }
    private:
    void timerCallback()
    {
        // Message for the string
        auto msg1 = std_msgs::msg::String();
        // String to send
        msg1.data = "Hello KUKA, hello BME!!!";
        // Publish string message on the topic
        first_pub_->publish(msg1);
    }
    // Publisher for string messages
    rclcpp::Publisher<std_msgs::msg::String>::SharedPtr first_pub_;
    // Timer to publish with desired rate
    rclcpp::TimerBase::SharedPtr timer_;
};

int main(int argc, char** argv)
{
    // Init ROS2
    rclcpp::init(argc, argv);
    // Create a node
    auto n = std::make_shared<MyPublisher>();
    rclcpp::spin(n);
    // Stop
    RCLCPP_INFO(n->get_logger(), "Stopping node.");
    rclcpp::shutdown();
    return 0;
}
```

Ebben a kódban ugyanaz maradt a node (`my_first_node`) és a topic (`my_topic`) neve. Ha a cpp fájl (`my_first_publisher.cpp`) nevén se változtattunk, akkor nincs szükség a `CMakeLists.txt`-ben bármit módosítani. (Ha más node nevet kívánunk használni, vagy a cpp fájl-t más néven mentettük el, akkor viszont szükséges a `CMakeLists.txt`-ben az eddigi bejegyzéseket módosítani, vagy újakat beszúrni.)

A workspace mappájában (`colcon_ws`) buildeljünk (`colcon build`). Futtassuk az új node-ot (`ros2 run my_pkg my_first_node`), és egy másik terminálban hallgassuk meg, hogy megjelennek-e az üzenetek a topicon (`ros2 topic echo /my_topic`).

ROS paraméter használata

Az új, saját osztályt használó `my_first_publisher.cpp` fájlban eddig az üzeneteket 2 másodpercenként (2000 milliszekundumonként) küldtük egy timer segítségével. Hozzunk most létre egy ROS paramétert, ami az üzenetek küldésének gyakoriságát határozza meg.

ROS2-ben a paraméterek a node-okhoz tartoznak, ezért a `MyPublisher()` konstruktorát módosítsuk az alábbiak szerint:

```
    MyPublisher() : Node("my_first_node") // Constructor of the node
    {
       // Display info for user
       RCLCPP_INFO(this->get_logger(), "First node with a parameter is
                                                              running.");
       // Using parameter (my_Ts: its name, 2000: its default value)
       this->declare_parameter("my_Ts", 2000);
       std::uint16_t my_Ts_param = this->get_parameter("my_Ts").as_int();
       RCLCPP_INFO(this->get_logger(), "Ts = %s ms.", std::to_string
                                                 (my_Ts_param).c_str());
       // Topic for a string
       first_pub_ = this->create_publisher<std_msgs::msg::String>
                                                       ("my_topic",1000);
       // Timer for sending
       timer_ = this->create_wall_timer(std::chrono::milliseconds
            (my_Ts_param), std::bind(&MyPublisher::timerCallback, this));
    }
```

A fenti kódban piros szín jelöli a változtatásokat. A `my_first_publisher.cpp` többi része változatlan marad. Buideljünk, és futtassuk a node-ot!

Egy másik terminálban ellenőrizhető, hogy létrejött-e a paraméter:

```
ros_user@ubuntu:~$ ros2 param list
```

Most változtassunk a parméter értékén:

```
ros_user@ubuntu:~$ ros2 param set my_first_node my_Ts 200
```

Azt tapasztaljuk, hogy az üzenetküldés gyakorisága nem változott. Ennek az oka az, hogy a paraméter értéket a node létrejöttekor kérdezzük le, után már nincs hatása. Állítsuk le a node-ot (`Ctrl`+`C`), majd indítsuk újra úgy, hogy más értéket adunk a paraméternek:

```
ros_user@ubuntu:~$ ros2 run my_pkg my_first_node --ros-args -p my_Ts:=10
```

Most már az új gyakorisággal küldi az üzeneteket.

Lehetőség van arra is, hogy a node paramétereinek értékét kiírjuk egy `yaml` kiterjesztésű fájlba. Ehhez hozzunk létre egy `config` mappát a package-en belül:

```
ros_user@ubuntu:~$ mkdir colcon_ws/src/my_pkg/config
ros_user@ubuntu:~$ ros2 param dump my_first_node > colcon_ws/src/my_pkg/
                                                  config/my_params.yaml
```

Megnézhetjük a `yaml` fájlt, és akár át is írhatjuk benne a `my_Ts` paraméter értékét. A node futtatható úgy, hogy a `yaml` fájlból töltjük be a paramétereit:

```
ros_user@ubuntu:~$ ros2 run my_pkg my_first_node --ros-args --params-file
                                colcon_ws/src/my_pkg/config/my_params.yaml
```

Első subscriber node implementálása

Most implementáljunk egy második node-ot, ami feliratkozik az első topic-jára. Legyen ez a **`my_first_subscriber.cpp`** fájl. (Továbbra is a `my_pkg/src` könyvtárba dolgozunk.)

```
   #include <memory>
```

```
#include "rclcpp/rclcpp.hpp"
#include "std_msgs/msg/string.hpp"
using std::placeholders::_1;

class MySubscriber : public rclcpp::Node
{
    public:
     MySubscriber() : Node("my_first_listener") // Constructor of the node
     {
        RCLCPP_INFO(this->get_logger(), "My first listener is running.");
        // Subscribing for the topic
        sub_ = this->create_subscription<std_msgs::msg::String>
   ("my_topic", 1000, std::bind(&MySubscriber::topicCallback, this, _1));
     }

    private:
     // Callback function to handle messages of the topic
     void topicCallback(const std_msgs::msg::String & msg)
     {
        // Display received message
        RCLCPP_INFO(this->get_logger(), "I heard: [%s]",
                                                    msg.data.c_str());
     }
     // Subscriber
     rclcpp::Subscription<std_msgs::msg::String>::SharedPtr sub_;
};
int main(int argc, char** argv)
{
    // Init ROS2
    rclcpp::init(argc, argv);
    // Create a node
    auto n = std::make_shared<MySubscriber>();
    rclcpp::spin(n);
    // Stop
    RCLCPP_INFO(n->get_logger(), "Stopping node.");
    rclcpp::shutdown();
    return 0;
}
```

A `my_pkg/CMakeLists.txt` állományt újra módosítani kell. Szúrjuk be a publisherhez (`my_first_node`) tartozó sorok alá a következőt:

```
add_executable(my_first_listener src/my_first_subscriber.cpp)
ament_target_dependencies(my_first_listener rclcpp std_msgs)
```

Illetve bővítsük ki a korábbi `install(TARGETS ...)`-t:

```
install(TARGETS
   my_first_node
   my_first_listener
   DESTINATION lib/${PROJECT_NAME})
```

Most jöhet a build (`colcon build`), és a node-ok futtatása külön terminálokban:

```
ros_user@ubuntu:~$ ros2 run my_pkg my_first_node
```

Illetve:

```
ros_user@ubuntu:~$ ros2 run my_pkg my_first_listener
```

Launch fájl létrehozása

Ha fárasztónak találjuk, hogy minden egyes node-hoz külön terminált kell nyitni, és egyesével kell őket futtatni, akkor létrehozhatunk egy launch fájt is. Ebben a node-ok indításán kívül egyéb műveleteket is el lehet végezni, például állítható a ROS paraméterek értéke is. Most írjunk egy launch fájt, ami futtatja a két node-unkat, és az üzenetküldés gyakoriságát is beállítja. A `my_pkg` könyvtárban hozzunk létre egy `launch` könyvtárt, abba pedig a **`first.launch.py`** állományt:

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

def generate_launch_description():
    return LaunchDescription([
        Node(
            package='my_pkg',
            executable='my_first_node',
            name='my_first_node',
            parameters=[{'my_Ts': 500}]
        ),
        Node(
            package='my_pkg',
            executable='my_first_listener',
            name='my_first_listener',
            output='screen',
            emulate_tty=True
        )
    ])
```

Buildelés után, egy terminálban az alábbi utasítással hívható meg a fenti launch fájl:

```
ros_user@ubuntu:~$ cd colcon_ws
ros_user@ubuntu:~/colcon_ws$ colcon build
ros_user@ubuntu:~/colcon_ws$ cd src/my_pkg/launch
ros_user@ubuntu:~/colcon_ws/src/my_pkg/launch$ ros2 launch
                                                    first.launch.py
```

Ha módosítani szeretnénk az üzenetküldés gyakoriságán, elegendő a launch fájlban átírni az értéket, menteni a fájlt, buildelni, és újra futtatni.

Lehetőség van arra is, hogy a `yaml` fájlból töltsük be a paramétereket. A `my_first_node` esetén a launch fájlban módosítsuk a `parameters`-t a következőképpen:

```
            parameters=['../config/my_params.yaml']
```

Buildelés után ellenőrizzük:

```
ros_user@ubuntu:~/colcon_ws/src/my_pkg/launch$ ros2 launch
                                                    first.launch.py
```

Futtatható a launch fájl úgy is, hogy az elérési út helyett a package nevét adjuk meg. Ehhez viszont módosítani kell a `CMakeLists.txt` fájlt. Az `install(TARGETS ...)` után szúrjuk be a következőt:

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

A launch fájlban pedig a `yaml` fájl elérési útvonalát a pirossal írt sorok segítségével lehet beállítani

```
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():
    return LaunchDescription([
        Node(
            package='my_pkg',
            executable='my_first_node',
            name='my_first_node',
            parameters= [os.path.join(get_package_share_directory
                            ('my_pkg'), 'config/my_params.yaml')]
        ),
        Node(
            package='my_pkg',
            executable='my_first_listener',
            name='my_first_listener',
            output='screen',
            emulate_tty=True
        )
    ])
```

Buildeljünk. Ezután már indítható a launch fájl az alábbi szintaktikával is:

```
ros_user@ubuntu:~$ ros2 launch my_pkg first.launch.py
```

Service használata

Most bonyolítsuk a `my_first_node` funkcióit! Legyen egy olyan service-e, amit, ha meghívnak, kicsit módosítja a szöveget, amit a `my_new_topic` nevű topic-ra küld. Ha újra meghívják ezt a service-t, akkor az eredeti szöveget küldi ugyanerre a topic-ra. Az alábbi fájl neve **`my_service_server.cpp`**.

```
#include <chrono>
#include "rclcpp/rclcpp.hpp"
#include "std_msgs/msg/string.hpp"
#include "std_srvs/srv/empty.hpp"
using namespace std::chrono_literals;
using namespace std::placeholders;

class MyServer : public rclcpp::Node
{
    public:
    MyServer() : Node("my_first_server") // Constructor of the node
    {
        // Display info for user
        RCLCPP_INFO(this->get_logger(), "First server is running.");
        // Using parameter my_Ts
       this->declare_parameter("my_Ts", 2000);
       std::uint16_t my_Ts_param = this->get_parameter("my_Ts").as_int();
       RCLCPP_INFO(this->get_logger(), "Ts = %s ms.", std::to_string
                                                  (my_Ts_param).c_str());
```

```
        // Topic for a string
        first_pub_ = this->create_publisher<std_msgs::msg::String>
                                                    ("my_new_topic", 1000);
        // Timer for sending
        timer_ = this->create_wall_timer(std::chrono::milliseconds
               (my_Ts_param), std::bind(&MyServer::timerCallback, this));
        // Service
        my_service_ = this->create_service<std_srvs::srv::Empty>
     ("my_service", std::bind(&MyServer::serviceCallback, this, _1, _2));

    }
    private:
    void timerCallback()
    {
        // Message for the string
        auto msg1 = std_msgs::msg::String();
        // String to send
        if(this->type_ == 1)
        {
            msg1.data = "Hello KUKA, hello BME!";
        }
        else
        {
            msg1.data = "Hello world!";
        }
        // Publish string message on the topic
        first_pub_->publish(msg1);
    }
    // Callback function to handle service calls
    void serviceCallback(const std::shared_ptr
                    <std_srvs::srv::Empty::Request> req, std::shared_ptr
                                    <std_srvs::srv::Empty::Response> res)
    {
        (void)req; // req and res are not used (they are empty)
        (void)res;
        if(this->type_ == 1) // Change of type_
        {
            this->type_ = 2;
        }
        else
        {
            this->type_ = 1;
        }
        RCLCPP_INFO(this->get_logger(), "my_service was called. Type is
                                                        %d", this->type_);
    }
    // Publisher for string messages
    rclcpp::Publisher<std_msgs::msg::String>::SharedPtr first_pub_;
    // Timer to publish with desired rate
    rclcpp::TimerBase::SharedPtr timer_;
    // Service server
    rclcpp::Service<std_srvs::srv::Empty>::SharedPtr my_service_;
    // Type of sending
    int type_ = 1;
```

```
};

int main(int argc, char** argv)
{
    // Init ROS2
    rclcpp::init(argc, argv);
    // Create a node
    auto n = std::make_shared<MyServer>();
    rclcpp::spin(n);
    // Stop
    RCLCPP_INFO(n->get_logger(), "Stopping node.");
    rclcpp::shutdown();
    return 0;
}
```

Új függőséget (`std_srvs`) kell hozzáadnunk a `package.xml`-hez és a `CMakeLists.txt`-hez. Utóbbiba az új node-nak megfelelő változtatásokat is el kell végzeni. Majd jöhetnek a szokásos lépések: `colcon build`, node futtatása (`ros2 run my_pkg my_first_server`). Egy új terminálban hallgassunk bele a `my_new_topic`-ba:

```
ros_user@ubuntu:~$ ros2 topic echo /my_new_topic
```

Egy sokadik terminálban pedig időnként hívjuk meg a `my_service` nevű service-t:

```
ros_user@ubuntu:~$ ros2 service call /my_service std_srvs/srv/Empty
```

Ha jól működik minden, akkor a `my_new_topic`-on megjelenő szöveg változik a `my_service` meghívásának függvényében.

Most a `my_first_listener` node kódjából (`my_first_subscriber.cpp`) kiindulva hozzuk létre a `my_client_node`-ot, ami egyrészt feliratkozik a `my_new_topic`-ra, másrészt időnként meghívja a `my_service` nevű service-t. Az állományunk neve lehet például **`my_service_client.cpp`**.

```
#include <memory>
#include<chrono>
#include "rclcpp/rclcpp.hpp"
#include "std_msgs/msg/string.hpp"
#include "std_srvs/srv/empty.hpp"
using std::placeholders::_1;
using namespace std::chrono_literals;

class MyClient : public rclcpp::Node
{
    public:
     MyClient() : Node("my_first_client") // Constructor
     {
        RCLCPP_INFO(this->get_logger(), "My first client is running.");
        // Service client
        client_ = this->create_client<std_srvs::srv::Empty>
                                                    ("my_service");
        // Subscribing for the topic
        sub_ = this->create_subscription<std_msgs::msg::String>
   ("my_new_topic", 1000, std::bind(&MyClient::topicCallback, this, _1));
     }

    private:
```

```
        // Callback funtion to handle messages of the topic
        void topicCallback(const std_msgs::msg::String & msg1)
        {
           // Display received message
           RCLCPP_INFO(this->get_logger(), "I heard: [%s]",
                                                    msg1.data.c_str());
           if(count_++ > 3) // Time to call the service
           {
               // Checking the availability of the service
               while(!client_->wait_for_service(1s))
               {
                   if(!rclcpp::ok())
                   {
                       RCLCPP_ERROR(this->get_logger(), "Exiting while
                                                waiting for the service.");
                       rclcpp::shutdown();
                   }
                   RCLCPP_INFO(this->get_logger(), "Waiting for service.");
               }
               // Empty request message
               auto my_req = std::make_shared
                                        <std_srvs::srv::Empty::Request>();
               // Service call
               client_->async_send_request(my_req);
               count_ = 1;
           }
        }
        // Subscriber
        rclcpp::Subscription<std_msgs::msg::String>::SharedPtr sub_;
        // Service client
        rclcpp::Client<std_srvs::srv::Empty>::SharedPtr client_;
        size_t count_ = 1;
    };

    int main(int argc, char** argv)
    {
        // Init ROS2
        rclcpp::init(argc, argv);
        // Create a node
        auto n = std::make_shared<MyClient>();
        rclcpp::spin(n);
        // Stop
        RCLCPP_INFO(n->get_logger(), "Stopping node.");
        rclcpp::shutdown();
        return 0;
    }
```

Szokásos lépéseink: `CMakeLists.txt` módosítása, `colcon build` és a szükséges node-ok futtatása. (Persze írhatunk egy újabb launch fájlt is, hogy ne kelljen egyesével futtatni a node-okat.)

ROS Bag használata

Ha egy topic üzeneteit későbbi felhasználásra el szeretnénk menteni, ROS bag-et használhatunk. Ez feliratkozik a topic-ra, és az elmentett üzeneteket később vissza lehet játszani róla. Így tudjuk egy topic üzeneteit menteni:

```
ros_user@ubuntu:~$ ros2 bag record -o my_saved_data /my_topic
```

Közben a `my_first_node`-dal generáljunk üzeneteket. Ha leállítottuk a bag fájlba mentést (`Ctrl+C`), akkor megjelenik az aktuális könyvtárban egy `my_saved_data` nevű mappa, amiben a mentett adatok találhatók (egy metaadatokat tartalmazó `yaml` fájl, és a felvett adatok a db3 állományban).

Így tudjuk visszajátszani az üzeneteket, amiket például a `my_first_listener` node-dal feldolgozhatunk:

```
ros_user@ubuntu:~$ ros2 bag play -l my_saved_data
```

A fenti utasítás automatikusan újrakezdi egy bag lejátszását, ha az adatok végére ért. Ha csak egyszer szeretnénk lejátszani a mentett adatokat, akkor a `-l` kapcsoló (loop) nélkül kell kiadni az utasítást.

Fájlok végső formája

Ha a fenti lépéseket végigcsináltuk, végül így néz ki a `CMakeLists.txt` állományunk:

```
cmake_minimum_required(VERSION 3.8)
project(my_pkg)

if(CMAKE_COMPILER_IS_GNUCXX OR CMAKE_CXX_COMPILER_ID MATCHES
                                                    "Clang")
  add_compile_options(-Wall -Wextra -Wpedantic)
endif()

# find dependencies
find_package(ament_cmake REQUIRED)
find_package(rclcpp REQUIRED)
find_package(std_msgs REQUIRED)
find_package(std_srvs REQUIRED)

add_executable(my_first_node src/my_first_publisher.cpp)
ament_target_dependencies(my_first_node rclcpp std_msgs)
add_executable(my_first_listener src/my_first_subscriber.cpp)
ament_target_dependencies(my_first_listener rclcpp std_msgs)
add_executable(my_first_server src/my_service_server.cpp)
ament_target_dependencies(my_first_server rclcpp std_msgs
                                                    std_srvs)
add_executable(my_first_client src/my_service_client.cpp)
ament_target_dependencies(my_first_client rclcpp std_msgs
                                                    std_srvs)

install(TARGETS
   my_first_node
   my_first_listener
   my_first_server
   my_first_client
   DESTINATION lib/${PROJECT_NAME})
```

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

if(BUILD_TESTING)
  find_package(ament_lint_auto REQUIRED)
  # the following line skips the linter which checks for
                                                    copyrights

  # comment the line when a copyright and license is added to
                                                 all source files
  set(ament_cmake_copyright_FOUND TRUE)
  # the following line skips cpplint (only works in a git repo)
  # comment the line when this package is in a git repo and
                                                          when
  # a copyright and license is added to all source files
  set(ament_cmake_cpplint_FOUND TRUE)
  ament_lint_auto_find_test_dependencies()
endif()

ament_package()
```

Ez pedig a `package.xml`:

```
<?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>my_pkg</name>
  <version>0.0.0</version>
  <description>TODO: Package description</description>
  <maintainer email="kuka@todo.todo">kuka</maintainer>
  <license>TODO: License declaration</license>

  <buildtool_depend>ament_cmake</buildtool_depend>

  <depend>rclcpp</depend>
  <depend>std_msgs</depend>
  <depend>std_srvs</depend>

  <test_depend>ament_lint_auto</test_depend>
  <test_depend>ament_lint_common</test_depend>

  <export>
    <build_type>ament_cmake</build_type>
  </export>
</package>
```