3. Pár egyszerű példa ROS1 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

2024

Bevezetés

Ebben a kiegészítő anyagban pár egyszerű példán keresztül mutatjuk be, hogy miként lehet ROS1 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 catkin munkaterünkben (`catkin_ws`). Ebben fogunk most dolgozni. (Vagy minden új terminál megnyitásakor be kell állítani, hogy ez az aktuális munkakörnyezet (`source devel/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 `my_pkg/src` könyvtárba hozzuk létre a **`my_first_publisher.cpp`** fájlt a következő kóddal:

```
#include "ros/ros.h"
#include "std_msgs/String.h"

int main(int argc, char** argv)
{
    // Init ROS
    ros::init(argc, argv, "my_first_node");
    // Create node handle
    ros::NodeHandle n;
    // Display info for user
    ROS_INFO("First node is running.");
    // Set publishing rate in Hz
    ros::Rate r(0.5);
    // Topic for a string
    ros::Publisher first_pub = n.advertise<std_msgs::String>("my_topic",
1000);
    // Loop for sending
    while(ros::ok())
    {
        // Message for the string
        std_msgs::String msg1;
        // String to send
        char my_word[] = "KUKA+BME";
        // Define data of the string message
        msg1.data = my_word;
        // Publish string message on the topic
        first_pub.publish(msg1);
        // Ensure desired rate
        r.sleep();
    }
}
```

Mielőtt build-elnénk a package-et, szükséges pár információt megadni a `my_pkg/CMakeLists.txt` állományban. Ehhez nyissuk meg, és keressük mega a `## Declare a C++ executable` szakaszt (nagyjából a 132. sor). Vegyük ki a `#` jelet az `add_executable( )` utasítás elől, és írjuk át a következőre:

```
add_executable(my_first_node src/my_first_publisher.cpp)
```

Továbbá a `## Specify libraries to link a library or executable target against` kezdetű szakasz (nagyjából 147. sor) után is kiszedhetjük a `#` jeleket, és írjuk át a mintát erre:

```
target_link_libraries(my_first_node
  ${catkin_LIBRARIES}
)
```

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

```
ros_user@ubuntu:~/catkin_ws/src$ cd ..
ros_user@ubuntu:~/catkin_ws$ catkin_make
```

Ha valamit elrontottunk, a kapott hibaüzenet alapján javítsuk! Amennyiben nincs hiba, futtassunk. Egy terminálban indítsunk el egy ROS Master-t:

```
ros_user@ubuntu:~$ roscore
```

Egy másik terminálban pedig az új node-ot:

```
ros_user@ubuntu:~$ rosrun 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 harmadik terminált, és nézzük meg, hogy létezik-e a topic, és ha igen, akkor mi van rajta:

```
ros_user@ubuntu:~$ rostopic list
ros_user@ubuntu:~$ rostopic echo /my_topic
```

Ha minden jól működik, leállíthatjuk a node-ot, és már nem kell a topic-ot se hallgatni, de a ROS Master fusson még.

ROS paraméter használata

A `my_first_publisher.cpp` fájlban eddig az üzenet küldésének gyakoriságát a `ros::Rate r(0.5);` utasítással állítottuk be. Most hozzunk létre egy ROS paramétert ehhez a frekvenciához:

```
ros_user@ubuntu:~$ rosparam set my_fr 0.25
```

Ellenőrizhető, hogy létrejött-e a paraméter:

```
ros_user@ubuntu:~$ rosparam list
```

A kódban pedig cseréljük le a megfelelő sort erre:

```
// Set publishing rate in Hz
float freq;
if (n.getParam("/my_fr",freq))
{
    ROS_INFO("Selected frequency is %f.", freq);
}
else
{
    freq = 0.5;
```

```
            ROS_WARN("Freqency is set to default (%f).", freq);
        }
    ros::Rate r(freq);
```

Egy build (`catkin_make`) után futtassuk újra a node-ot. (Ha véletlenül leállítottuk a Mastert, akkor egy `roscore`-ral újra kell indítani.)

```
ros_user@ubuntu:~$ rosrun my_pkg my_first_node
```

Egy másik terminálban továbbra is figyeljük az üzeneteket a topic-on:

```
ros_user@ubuntu:~$ rostopic echo /my_topic
```

Állítsuk le a `my_first_node`-ot, írjuk át a frekvenciát, és futtassuk újra:

```
ros_user@ubuntu:~$ rosparam set my_fr 1.0
ros_user@ubuntu:~$ rosrun my_pkg my_first_node
```

Ha minden jól sikerült, akkor most már gyakrabban küldi a node az üzenetet. Próbáljuk ki azt is, hogy mi történik, ha véletlenül nem létezik a ROS paraméter. Töröljük ki a paramétert, és utána futtassuk a node-ot:

```
ros_user@ubuntu:~$ rosparam delete my_fr
ros_user@ubuntu:~$ rosrun my_pkg my_first_node
```

Kapunk egy sárgával megjelenő Warning-ot, és a default 0.5 Hz-es gyakorisággal történik az üzenetküldés.

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 "ros/ros.h"
#include "std_msgs/String.h"

// Callback for string topic – what to do if we get a message
void topicCallback(const std_msgs::String::ConstPtr& msg)
{
    ROS_INFO("I heard: [%s]", msg->data.c_str());
}

int main(int argc, char** argv)
{
    // Init ROS
    ros::init(argc, argv, "my_first_listener");
    // Create node handle
    ros::NodeHandle n;
    // Subscribes for the string topic
    ros::Subscriber sub = n.subscribe("my_topic",1000,topicCallback);
    // Enter loop, to call callback
    ros::spin();
}
```

A `my_pkg/CMakeLists.txt` állományba újabb két bejegyzést kell hozzáadni a korábbiak folytatásaként:

```
add_executable(my_first_listener src/my_first_subscriber.cpp)
```

Illetve:

```
target_link_libraries(my_first_listener
  ${catkin_LIBRARIES}
)
```

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

```
ros_user@ubuntu:~$ rosrun my_pkg my_first_node
```

Illetve:

```
ros_user@ubuntu:~$ rosrun 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 megnyitja a két node-unkat és az üzenetküldési frekvenciát is beállítja. A `my_pkg` könyvtárban hozzunk létre egy `launch` könyvtárt, abba pedig a **`first.launch`** állományt:

```
<launch>
    <node pkg="my_pkg" name="node1" type="my_first_node"/>
    <node pkg="my_pkg" name="node2" type="my_first_listener"
output="screen"/>
    <param name=" my_fr " type="double" value="1.0" />
</launch>
```

Egy terminálban az alábbi utasítással hívható meg a fenti launch fájl:

```
ros_user@ubuntu:~$ roslaunch my_pkg first.launch
```

Mivel a `my_first_listener` node-nál azt adtuk meg, hogy a kimenet jelenjen meg a terminálban (`output="screen"`), ennek a node-nak az üzeneteit olvashatjuk futás közben. (Ha szeretnénk a `my_first_node` üzenetei is megjeleníthetők hasonlóan.)

Ha módosítani szeretnénk az üzenetküldés frekvenciáján, elegendő a launch fájlban átírni az értéket, menteni a fájlt, és a fenti sorral újra megnyitni.

Módosítsuk a publisher-t

Most csináljunk egy bonyolultabb publisher node-ot. Ne egy állandó szöveget küldjünk el, hanem a felhasználó által megadottat. Ezt az üzenetet nem rendszeresen fogjuk kiküldeni, hanem olyan gyakorisággal, ahogy az új szövegek érkeznek. Az új node kódját a **`my_second_publisher.cpp`** fájlba írjuk.

```
#include "ros/ros.h"
#include "std_msgs/String.h"

int main(int argc, char** argv)
{
    // Init ROS
    ros::init(argc, argv, "my_clever_node");
    // Create node handle
    ros::NodeHandle n;
    // Display info for user
    ROS_INFO("Clever node is running!");
```

```
    // Topic for a string
    ros::Publisher first_pub = n.advertise<std_msgs::String>("my_topic",
1000);
    // Loop for sending
    while(ros::ok())
    {
        // Message for the string
        std_msgs::String msg1;
        // String to send
        char my_word[20];
        printf("Type a word or 0 for exit: \n");
        scanf("%20s", my_word);

        if(my_word[0]=='0')
        {
            // Exit the node
            ros::shutdown();
        }
        else
        {
            // Define data of the string message
            msg1.data = my_word;
            // Publish string message on the topic
            first_pub.publish(msg1);
        }
    }
}
```

Jöhetnek a szokásos lépések: Végezzük el a my_pkg/CMakeLists.txt állományban a szükséges sorok hozzáadását:

```
add_executable(my_clever_node src/my_second_publisher.cpp)
```

Illetve:

```
target_link_libraries(my_clever_node
  ${catkin_LIBRARIES}
)
```

Build-eljünk (`catkin_make`) és futtassuk az új node-ot :

```
ros_user@ubuntu:~$ rosrun my_pkg my_clever_node
```

Ha szavakat adunk meg, azokat a `my_topic`-ra küldi ez a node is. Így, ha egy külön terminálban újra elindítjuk a `my_first_listener` node-ot, akkor ezeket az üzeneteket is megkapja:

```
ros_user@ubuntu:~$ rosrun my_pkg my_first_listener
```

Ha nem szeretnénk az üzenetek keveredését, akkor nevezzük át a topic-ot, amire a `my_clever_node` publish-ol a `my_second_publisher.cpp` kódban, és ezt a topic nevet használjuk ott, ahol szükség van rá. (Például írhatunk egy `my_second_subscriber.cpp`-t.)

Service használata

Most bonyolítsuk a `my_clever_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, a felhasználó által beírt szót küldi ugyanerre a topic-ra. Mivel most már két dolgot

kell párhuzamosan csinálni (publish-olni egy topic-ra, reagálni a service hívásra) thread-eket használunk. Az alábbi fájl neve **my_service_server.cpp**.

```
#include "ros/ros.h"
#include "std_msgs/String.h"
#include "std_srvs/Empty.h"
#include "thread"

ros::Publisher pub;
ros::ServiceServer first_service;
int type = 1;
// Callback for service call
bool s_callback(std_srvs::Empty::Request& request,
std_srvs::Empty::Response& response)
{
    if(type == 1)
    {
        type = 2;
    }
    else
    {
        type = 1;
    }
    ROS_INFO("Service called, type has changed.");
    return true;
}
// Creating message to send
void createMessage()
{
    while(ros::ok())
    {
        // Message for the string
        std_msgs::String msg;
        char my_word[20];
        char string_to_send[35];
        printf("Type a word or 0 for exit: \n");
        scanf("%20s", my_word);
        if(my_word[0]=='0')
        {
            ros::shutdown();
        }
        else
        {
            strcpy(string_to_send, my_word);
            if(type == 2)
            {
                strcat(string_to_send, " from KUKA + BME");
            }
            msg.data = string_to_send;
            pub.publish(msg);
        }
    }
}
```

```
int main(int argc, char** argv)
{
    // Init ROS
    ros::init(argc, argv, "my_server_node");
    // Create node handle
    ros::NodeHandle n;
    // Display info for user
    ROS_INFO("Server node is running!");
    // Topic for a string
    pub = n.advertise<std_msgs::String>("my_new_topic", 1000);
    // Advertising a service
    first_service = n.advertiseService("my_service", s_callback);
    // New thread for the publishing
    std::thread worker(createMessage);
    // Ensure continuous running
    ros::spin();
    worker.join();
}
```

Jöhetnek a szokásos lépések: `CMakeLists.txt` módosítása, `catkin_make`, node futtatása (`rosrun my_pkg my_server_node`). Egy új terminálban hallgassunk bele a `my_new_topic`-ba:

```
ros_user@ubuntu:~$ rostopic echo /my_new_topic
```

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

```
ros_user@ubuntu:~$ rosservice call /my_service
```

Ha jól működik minden, akkor a `my_server_node`-nak begépelt szöveg megjelenik a `my_new_topic`-on, de a `my_service` meghívásának függvényében néha kis kiegészítést kap.

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 "ros/ros.h"
#include "std_msgs/String.h"
#include "std_srvs/Empty.h"

std_srvs::Empty my_srv;
int count = 0;
// Callback for string topic
void topicCallback(const std_msgs::String::ConstPtr& msg)
{
    ROS_INFO("I heard: [%s]", msg->data.c_str());
    count++;
    if(count == 3)
    {
        // Calling the service
        ros::service::call("my_service", my_srv);
        ROS_INFO("Service called.");
        count = 0;
    }
}

int main(int argc, char** argv)
```

```
{
    // Init ROS
    ros::init(argc, argv, "my_client_node");
    // Create node handle
    ros::NodeHandle n;
    // Subscribes for the string topic
    ros::Subscriber sub = n.subscribe("my_new_topic",1000,topicCallback);
    // Enter loop, to call callback
    ros::spin();
}
```

Szokásos lépéseink: `CMakeLists.txt` módosítása, `catkin_make` 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:~$ rosbag record /my_topic
```

Közben a `my_clever` node-dal generáljunk üzeneteket. Ha leállítottuk a bag fájlba mentést, akkor megjelenik a `catkin_ws` mappában ben egy `.bag` kiterjesztésű fájl (pl. `2024-09-01-23-59-59.bag`).

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

```
ros_user@ubuntu:~$ rosbag play 2024-09-01-23-59-59.bag
```