UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

Desarrollo y Gestión de Interfaces Personalizadas en ROS2

Eugenio Ivorra

En esta práctica, se pretende explorar en profundidad la creación, implementación y gestión de interfaces personalizadas en ROS2. Este conocimiento permitirá una integración más efectiva de funciones personalizadas y una manipulación más flexible de la configuración en tiempo de ejecución en ROS2. Los objetivos específicos de este módulo son:

- Entender los antecedentes y la necesidad de interfaces personalizadas en la comunicación entre nodos de ROS2.
- Familiarizarse con el proceso de creación de un nuevo paquete en ROS2 y aprender a definir mensajes, servicios y acciones personalizados para el mismo.
- Aprender a actualizar los archivos CMakeLists.txt y package.xml para incluir y gestionar las nuevas interfaces personalizadas.
- Adquirir habilidades prácticas en la implementación de nodos de cliente y servidor en Python que utilicen estas interfaces personalizadas.

Índice general

1 Interfaces personalizadas

1.1 Introducción

El objetivo de esta práctica es definir interfaces personalizadas (`.msg`, `.srv` y `.action`) y usarlas con nodos Python en ROS 2. En prácticas previas, se utilizaron interfaces de mensajes y servicios para aprender sobre tópicos, servicios y nodos de editor/subscriptor y servicio/cliente en Python pero estas interfaces estaban ya predefinidas. Aunque es buena práctica usar definiciones de interfaz predefinidas para hacer nuestro código más compatible con terceros, en ocasiones también necesitarás definir tus propios mensajes, servicios y acciones para tener un código más legible y óptimo. Esta práctica te introducirá a la creación de estas definiciones de interfaz personalizadas.

1.2 Creación de un Nuevo Paquete

Para esta práctica, crearás archivos `.msg`, `.srv` y `.action` personalizados en su propio paquete y luego los utilizarás en un paquete separado. Ambos paquetes deben estar en el mismo espacio de trabajo.

```
ros2 pkg create --build-type ament_cmake tutorial_interfaces
```

`tutorial_interfaces` es el nombre del nuevo paquete. Aunque es un paquete CMake, esto no limita en qué tipo de paquetes puedes usar tus mensajes, servicios y acciones.

1.3 Definición de Mensajes, Servicios y Acciones Personalizadas

Dentro de tu paquete, crea directorios para tus archivos de mensaje, servicios y acciones con los siguientes comandos:

```
mkdir -p ros2_ws/src/tutorial_interfaces/msg
mkdir -p ros2_ws/src/tutorial_interfaces/srv
mkdir -p ros2_ws/src/tutorial_interfaces/action
```

1.3.1 Mensajes

En el directorio `tutorial_interfaces/msg`, crea un archivo llamado `Num.msg`: Que contendrá un número.

```
int64 num
```

Y otro llamado `Sphere.msg` en el que se combina un float con un tipo de variable de otro paquete, en concreto un tipo punto del paquete geometry_msgs:

```
geometry_msgs/Point center
float64 radius
```

1.3.2 Servicios

Dentro de `tutorial_interfaces/srv`, crea un archivo llamado `AddThreeInts.srv`:

```
int64 a
int64 b
int64 c
---
int64 sum
```

Recuerda que los guiones separan los argumentos de la petición de la respuesta del servidor.

1.3.3 Acciones

Para definir una acción personalizada, hay que crear un archivo `.action` en un directorio llamado `action`. Dentro de dicho directorio, crea un archivo llamado `ComputeSum.action`:

```
int64 a
int64 b
int64 c
---
int64 sum
---
float64 percent_complete
```

Los tres primeros números definen la petición (goal). El siguiente número la respuesta final de la acción (result). Y el último número la retroalimentación que proporciona (feedback).

1.4 Actualización de CMakeLists.txt y package.xml

Para generar código específico del lenguaje a partir de tus interfaces, debes actualizar `CMakeLists.txt` y `package.xml`.

1.4.1 CMakeLists.txt

Añade las siguientes líneas en el fichero CMakeList justo antes de la línea `ament_package()` para poder compilar las nuevas interfaces y que tengan las dependencias correctas (`geometry_msgs`):

```
find_package(rosidl_default_generators REQUIRED)
find_package(geometry_msgs REQUIRED)

rosidl_generate_interfaces(${PROJECT_NAME}
"msg/Num.msg"
"msg/Sphere.msg"
"srv/AddThreeInts.srv"
"action/ComputeSum.action"
DEPENDENCIES geometry_msgs
)
```

Estas líneas en el archivo `CMakeLists.txt` de un paquete ROS2 están relacionadas con la compilación de interfaces personalizadas, que pueden incluir mensajes, servicios y acciones. A continuación, se desglosa cada línea y su propósito:

1. `find_package(rosidl_default_generators REQUIRED)`:
   - Esta línea busca e incluye el paquete `rosidl_default_generators` en el proceso de compilación.
   - `rosidl_default_generators` es un paquete ROS2 que proporciona herramientas para generar código de lenguaje específico (por ejemplo, Python, C++, etc.) a partir de las definiciones de las interfaces (mensajes, servicios y acciones).
   - Es esencial incluir este paquete para convertir las definiciones de las interfaces en código que pueda ser utilizado por otros nodos y paquetes en ROS2.
2. `find_package(geometry_msgs REQUIRED)`:
   - Esta línea busca e incluye el paquete `geometry_msgs`.
   - `geometry_msgs` es un paquete estándar en ROS que contiene definiciones de mensajes para representar formas y patrones geométricos comunes.
3. `rosidl_generate_interfaces(${PROJECT_NAME} ... DEPENDENCIES geometry_msgs)`:
   - Esta función `rosidl_generate_interfaces` se encarga de generar el código necesario para las interfaces (mensajes, servicios y acciones) definidas en este paquete.

- `${PROJECT_NAME}` es una variable que representa el nombre del paquete actual. Es una forma estándar de referirse al paquete actual dentro del archivo `CMakeLists.txt`.

- Las cadenas de texto, como `"msg/Num.msg"`, `"msg/Sphere.msg"`, etc., son las rutas relativas a las definiciones de las interfaces dentro del paquete. En este caso, el paquete tiene definiciones para dos mensajes (`Num` y `Sphere`), un servicio (`AddThreeInts`) y una acción (`ComputeSum`).

- La opción `DEPENDENCIES` indica las dependencias externas de estas interfaces. En este caso, tenemos solo la de `geometry_msgs`.

1.4.2 package.xml

Añade las siguientes líneas en el fichero package antes de la sección de `export`. Fíjate que aquí también es necesario especificar la dependencia a (`geometry_msgs`). El resto de líneas se encargan de compilar y enlazar las interfaces con el entorno Python.

```
<depend>geometry_msgs</depend>

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

1.5 Compilar y probar

Una vez definidas tus interfaces, puedes compilar tu paquete (en la raiz de tu workspace) y probarlas:

```
colcon build --packages-select tutorial_interfaces
source install/setup.bash
```

Después de compilar (es normal que la primera vez que compilas le cueste un rato), puedes usar las herramientas de ROS 2 para inspeccionar tus interfaces:

```
ros2 interface show tutorial_interfaces/msg/Num
ros2 interface show tutorial_interfaces/srv/AddThreeInts
ros2 interface show tutorial_interfaces/action/ComputeSum
```

Para ver todas las interfaces disponibles podemos usar:

```
ros2 interface list
```

1.6 Implementación de Nodos de Cliente y Servidor en Python

En esta sección, te guiaré a través de la implementación de nodos de cliente y servidor en Python para ROS 2 utilizando las interfaces personalizadas que hemos creado con el ejemplo concreto de la interfaz de servicio. El resto de interfaces se utilizan de forma similar.

1.6.1 Nodo Servidor

Vamos a implementar un servidor que utiliza el servicio `AddThreeInts.srv`. El servidor recibirá tres números enteros, los sumará y devolverá la suma como respuesta. Crea un archivo llamado `add_three_ints_server.py` en tu paquete `mi_paquete_python`:

```
import rclpy
from rclpy.node import Node
from tutorial_interfaces.srv import AddThreeInts

class AddThreeIntsServer(Node):
    def __init__(self):
        super().__init__('add_three_ints_server')
        self.srv = self.create_service(AddThreeInts, 'add_three_ints',
        ↪  self.add_callback)

    def add_callback(self, request, response):
        response.sum = request.a + request.b + request.c
        self.get_logger().info(f"Incoming request: a: {request.a}, b:
        ↪  {request.b}, c: {request.c}")
        self.get_logger().info(f"Sending back response: {response.sum}")
        return response
def main(args=None):
    rclpy.init(args=args)
    server = AddThreeIntsServer()
    rclpy.spin(server)
    server.destroy_node()
    rclpy.shutdown()

if __name__ == '__main__':
    main()
```

1.6.2 Nodo Cliente

A continuación, crearemos un nodo cliente que enviará una solicitud al servidor. Crea un archivo llamado `add_three_ints_client.py` en tu paquete `mi_paquete_python`:

```
import rclpy
from rclpy.node import Node
from tutorial_interfaces.srv import AddThreeInts
import sys
```

```
class AddThreeIntsClient(Node):
    def __init__(self, a, b, c):
        super().__init__('add_three_ints_client')
        self.cli = self.create_client(AddThreeInts, 'add_three_ints')
        while not self.cli.wait_for_service(timeout_sec=1.0):
            self.get_logger().info('service not available, waiting again...')
        self.req = AddThreeInts.Request()
        self.req.a = a
        self.req.b = b
        self.req.c = c

    def send_request(self):
        future = self.cli.call_async(self.req)
        future.add_done_callback(self.future_callback)

    def future_callback(self, future):
        try:
            response = future.result()
            self.get_logger().info(f"Sum: {response.sum}")
            # Shutdown rclpy to make the node terminate
            rclpy.shutdown()
        except Exception as e:
            self.get_logger().error(f"Service call failed {e}")

def main(args=None):
    rclpy.init(args=args)
    if len(sys.argv) < 4:
        print("Usage: script_name.py <a> <b> <c>")
        return

    a = int(sys.argv[1])
    b = int(sys.argv[2])
    c = int(sys.argv[3])

    client = AddThreeIntsClient(a, b, c)
    client.send_request()
    rclpy.spin(client)
    client.destroy_node()

if __name__ == '__main__':
    main()
```

1.6.3 Configuración de setup.py y package.xml

Edita el archivo `setup.py` con el siguiente contenido para añadir los puntos de entrada:

```
entry_points={
    'console_scripts': [
        'add_three_ints_server =
        ↪  mi_paquete_python.add_three_ints_server:main',
        'add_three_ints_client =
        ↪  mi_paquete_python.add_three_ints_client:main'
    ],
},
```

Además añade la nueva dependencia a tu fichero `package.xml`:

```
<depend>tutorial_interfaces</depend>
```

1.6.4 Prueba de los Nodos

Una vez implementados los nodos, es momento de probarlos. Para ello necesitas compilar tu espacio de trabajo y cargar las configuraciones.

```
colcon build --packages-select mi_paquete_python
source install/setup.bash
```

Después de la compilación y de cargar el archivo de configuración, deberías ser capaz de ejecutar tus nodos usando `ros2 run`.

Inicia primero el nodo servidor:

```
ros2 run mi_paquete_python add_three_ints_server
```

Y en una nueva terminal, inicia el nodo cliente:

```
ros2 run mi_paquete_python add_three_ints_client 10 20 30
```

Observa los logs tanto en la terminal del cliente como en la del servidor, donde podrás ver la solicitud y la respuesta.

1.7 Actividad

Modifica tu servidor de acciones de la práctica anterior para que el goal incluya un punto (`geometry_msgs/Point`), una velocidad lineal ((float64)) y el radio ((float64)). El resto de la petición quedaría igual. El servidor de acciones tendrá que dibujar un círculo con la velocidad lineal solicitada, el radio y empezando en el punto que se le pasa. Para situar la tortuga en este punto utiliza el servicio `turtle1/teleport_absolute`.

Bibliografía

Web de los tutoriales de ROS (2023). https://docs.ros.org/en/humble/Tutorials.html.

Web de ROS2 Humble (2023). https://docs.ros.org/en/humble/index.html.