# ROS2 Apuntes Avanzado

**Por qué y cuándo usar ROS2 actions** ..... **2**
**Cómo funcionan las actions en ROS2** ..... **2**
**Creando un “Action Definition”** ..... **4**
**Action server en C++** ..... **4**
Uso de rclcpp::Rate ..... 7
**Action client en C++** ..... **7**

# Por qué y cuándo usar ROS2 actions

Primero, echemos un vistazo a cómo funcionan las cosas con la comunicación en ROS2, centrándonos en los temas y los servicios. Los temas son como canales de mensajes entre nodos, mientras que los servicios son como una especie de charla de ida y vuelta entre un cliente y un servidor. El cliente hace una solicitud, y el servidor responde con lo que se le pide.

Ahora bien, a medida que las aplicaciones robóticas evolucionan, nos hemos dado cuenta de que la comunicación a través de servicios tiene sus límites. Y es aquí donde entran en juego las acciones.

Las acciones en ROS2 son la solución cuando la comunicación cliente-servidor lleva tiempo, como cuando necesitas mover una parte física de un robot en el espacio. Mientras que los servicios son un poco asíncronos, las acciones son geniales para manejar tareas que toman su tiempo sin dejar al cliente colgado. Esto es clave cuando necesitas cancelar una tarea en marcha o recibir actualizaciones mientras se está ejecutando.

La gran diferencia entre servicios y acciones es que las acciones son expertas en manejar tareas que toman tiempo y requieren interacciones más complejas.

Un problema que las acciones resuelven es gestionar múltiples solicitudes al mismo tiempo. A diferencia de los servicios, las acciones son eficientes para manejar varias solicitudes, seleccionar entre ellas y lidiar con situaciones en las que necesitas reemplazar o cancelar una acción sin fastidiar a las demás.

Resumiendo, los temas son ideales para pasar datos de un lado a otro, los servicios son buenos para charlas rápidas entre cliente y servidor, pero cuando la tarea lleva su tiempo y necesitas cosas como cancelaciones y actualizaciones, las acciones son la opción a utilizar.

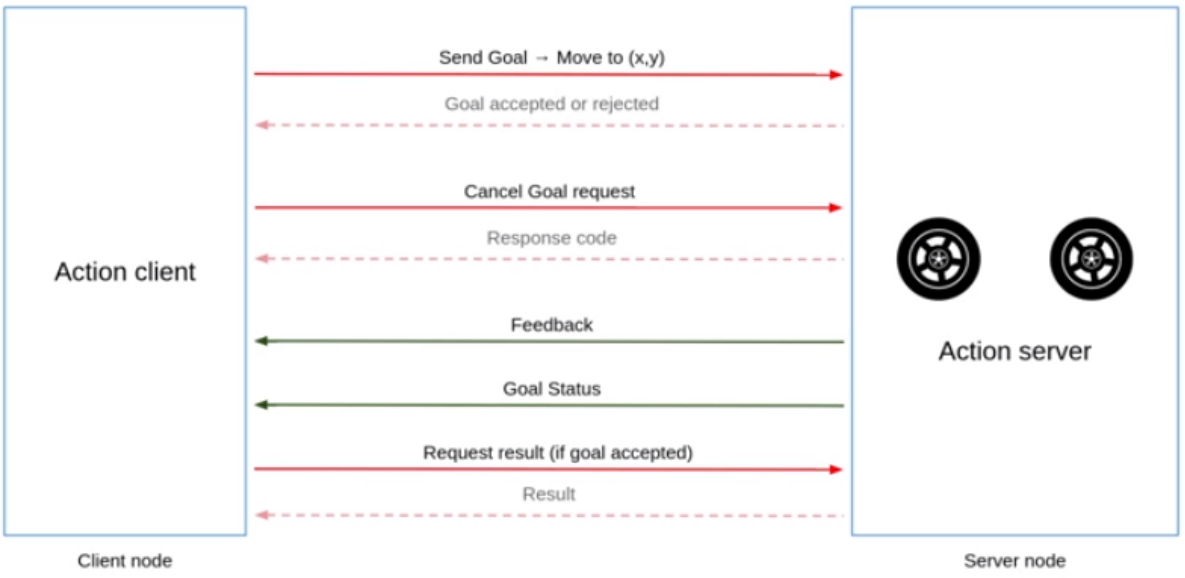
# Cómo funcionan las actions en ROS2

Imagina a la izquierda nuestro nodo cliente, equipado con un "action client", y a la derecha el nodo servidor, que alberga el "action server" y controla las ruedas del robot. ¿Cómo se desarrolla la ejecución de una acción?

En primer lugar, el servidor debe recibir una instrucción. El cliente inicia el proceso enviando un “goal” (objetivo) al servidor, algo parecido a una solicitud en términos de servicios. Si el servidor acepta el objetivo, el proceso continúa; de lo contrario, el cliente recibe una respuesta de rechazo, y la historia llega a su fin.

En caso de aceptación, el servidor procesa el objetivo ejecutando las acciones necesarias. Al mismo tiempo, el cliente envía una solicitud de resultado, esperando pacientemente a que el servidor complete la ejecución. Una vez finalizado, el servidor envía el resultado de vuelta al cliente, cerrando el círculo.

Este es el núcleo mínimo para un servidor de acción: recibir un objetivo y responder con un resultado. Pero las acciones en ROS2 ofrecen funcionalidades adicionales. Pueden incluir retroalimentación, que permite al servidor enviar información al cliente durante la ejecución, y un mecanismo de cancelación para interrumpir la ejecución del objetivo. Estos elementos, aunque opcionales, mejoran significativamente la versatilidad de las acciones.




Imágen 1.1 - Ejemplo action client y action server

En la implementación usada como ejemplo, estas funcionalidades se traducen en tres servicios (en rojo en la imagen 1.1) y dos topics (en verde) asociados a una acción. Los servicios gestionan el envío de objetivos, la cancelación y la obtención de resultados, mientras que los topics facilitan la comunicación de retroalimentación y el estado del objetivo. La buena noticia es que las clases de client y server action en ROS2 ya implementan estos servicios y topics.

Es importante señalar que las acciones admiten la posibilidad de varios clientes. Mientras que un “server action” es único para cada acción, varios nodos clientes

pueden enviar objetivos, y el servidor gestionará todo de manera eficiente. Un cliente puede enviar varios “goal” sin problemas.

# Creando un “Action Definition”

Si ya se sabe crear “.msg” o .”srv” crear “.action” es el mismo proceso. Si hay dudas, ver los apuntes de ros2_comienzo y buscar el apartado de interfaces. Hay que configurar los archivos CMakeList.txt y package.xml exactamente igual que para los msg y srv.

En el paquete dedicado a las interfaces, creamos una nueva carpeta llamada “action” donde incluiremos todos los ficheros “.action”. El nombre de los ficheros sigue la convención de PascalCase. A continuación se muestra un ejemplo de una interfaz “.action”

“CountUntil.action”

```
# Goal
int64 target_number
float64 period
---
# Result
int64 reached_number
---
# Feedback (opcional)
int64 current_number
```

# Action server en C++

A continuación se explica la creación de un servidor de acciones en ROS2 detallando las partes del código que no se hayan visto en los apuntes anteriores. Como ejemplo se utilizan los ficheros “count_until_server.hpp” y “count_until_server.cpp”.

“count_until_server.hpp”

**Creación del action server:**

`rclcpp_action::Server<CountUntil>::SharedPtr count_until_server_{};`: Esta es la declaración de la variable miembro `count_until_server_`. Es un puntero compartido (`SharedPtr`) a un objeto del tipo `rclcpp_action::Server<CountUntil>`.

`count_until_server_ = rclcpp_action::create_server<CountUntil>(...)`: Esta línea está creando un servidor de acciones. El servidor de acciones se está creando para manejar acciones del tipo `CountUntil`. `CountUntil`es la interfaz de acción creada anteriormente.

`this, "count_until", ...`: El primer argumento para `create_server` es el propietario del servidor (en este caso, `this`), y el segundo argumento es el nombre del servidor de acciones.

`std::bind(&CountUntilServerNode::goal_callback, this, std::placeholders::_1, std::placeholders::_2), ...`: Aquí se está vinculando la función `goal_callback` de la clase `CountUntilServerNode` a ser el callback de la meta para el servidor de acciones. Cuando se recibe una nueva meta, se llamará a esta función.

`std::bind(&CountUntilServerNode::cancel_callback, this, std::placeholders::_1), ...`: Similar a la línea anterior, esta línea vincula la función `cancel_callback` de la clase `CountUntilServerNode` a ser el callback de cancelación para el servidor de acciones. Esta función se llamará si se solicita la cancelación de una meta.

`std::bind(&CountUntilServerNode::handle_accepted_callback, this, std::placeholders::_1))`: Esta línea vincula la función `handle_accepted_callback` de la clase `CountUntilServerNode` a ser el callback para manejar metas aceptadas. Esta función se llamará cuando una meta haya sido aceptada por el servidor de acciones.

**Declaración de los métodos usados en el server action:**

rclcpp_action::GoalResponse goal_callback(const rclcpp_action::GoalUUID &uuid, std::shared_ptr<const CountUntil::Goal> goal);: Este método es un callback que se invoca cuando se recibe una nueva meta desde un cliente. Toma dos parámetros: un UUID de la meta y un puntero compartido a la meta. El UUID es un identificador único para la meta. La meta es de tipo CountUntil::Goal. El método devuelve una respuesta de tipo rclcpp_action::GoalResponse, que indica si la meta debe ser aceptada o rechazada.

rclcpp_action::CancelResponse cancel_callback(const std::shared_ptr<rclcpp_action::ServerGoalHandle<CountUntil>> goal_handle);: Este método es un callback que se invoca cuando se recibe una solicitud para cancelar

una meta. Toma un parámetro: un puntero compartido a un ServerGoalHandle, que es un objeto que permite al servidor de acciones interactuar con la meta. El método devuelve una respuesta de tipo rclcpp_action::CancelResponse, que indica si la solicitud de cancelación debe ser aceptada o rechazada.

### void handle_accepted_callback(const std::shared_ptr<rclcpp_action::ServerGoalHandle<CountUntil>> goal_handle);

: Este método es un callback que se invoca cuando una meta ha sido aceptada. Toma un parámetro: un puntero compartido a un ServerGoalHandle. No devuelve nada.

### void execute_goal(const std::shared_ptr<rclcpp_action::ServerGoalHandle<CountUntil>> goal_handle)

;: Este método es responsable de ejecutar la meta. Toma un parámetro: un puntero compartido a un ServerGoalHandle. No devuelve nada. Este método contendrá la lógica para realizar la tarea especificada por la meta.

---

En el fichero “count_until_server.cpp“, destacar lo siguiente:

**return de las funciones miembro**

### rclcpp_action::GoalResponse::ACCEPT_AND_EXECUTE

es una enumeración que se utiliza para determinar cómo debe responder el servidor de acciones a una solicitud de objetivo entrante.

En este caso, ACCEPT_AND_EXECUTE indica que el servidor de acciones ha aceptado el objetivo y comenzará a ejecutarlo inmediatamente. Esto es típicamente parte de una función de devolución de llamada que se invoca cuando se recibe una nueva solicitud de objetivo.

### rclcpp_action::CancelResponse::ACCEPT

es una enumeración que se utiliza para determinar cómo debe responder el servidor de acciones a una solicitud de cancelación de objetivo.

En este caso, ACCEPT indica que el servidor de acciones ha aceptado la solicitud de cancelación y cancelará la ejecución del objetivo. Esto es típicamente parte de una función de devolución de llamada que se invoca cuando se recibe una nueva solicitud de cancelación.

### Uso de rclcpp::Rate

**rclcpp::Rate** es una clase en ROS2 que se utiliza para hacer que un bucle se ejecute a una frecuencia específica. El constructor de Rate toma un argumento que es la frecuencia deseada en Hz (ciclos por segundo).

En este caso, 1.0/period calcula la frecuencia en Hz a partir de un período dado. Por ejemplo, si el período es de 2 segundos, entonces la frecuencia es de 0.5 Hz, lo que significa que el bucle se ejecutará una vez cada 2 segundos.

Aquí hay un ejemplo de cómo podría verse en un contexto más amplio:

```
double period = 2.0; // period in seconds
rclcpp::Rate loop_rate(1.0/period);

while (rclcpp::ok()) {
  // do some work here
  loop_rate.sleep();
}
```

En este ejemplo, se crea un objeto Rate con una frecuencia de 0.5 Hz. Luego, en el bucle, se hace algún trabajo y luego se llama a loop_rate.sleep(). Esta llamada a sleep hará que el bucle se detenga el tiempo suficiente para asegurar que la próxima iteración no comience hasta que hayan pasado 2 segundos desde el inicio de la iteración actual.

## Action client en C++

Mismo proceso que con los servers, pero ahora se utilizan como ejemplo los ficheros “count_until_client.hpp” y “count_until_client.cpp”