UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

Comunicación entre Nodos en ROS2: Implementación y Manipulación de Acciones y Clientes

Eugenio Ivorra

En esta práctica, se busca ampliar la comprensión sobre la comunicación entre nodos en ROS2, mediante la implementación y manipulación de servidores de Acciones y clientes en Python, y aplicar este conocimiento en un contexto práctico utilizando el simulador turtlesim. Los objetivos específicos de esta práctica incluyen:

- Profundizar en el entendimiento sobre la estructura y funcionamiento de las Acciones en ROS2, y cómo estas se diferencian de los servicios y del modelo de publicador-suscriptor.
- Comprender la estructura de un nodo servidor de Acciones en Python que pueda recibir peticiones usando una interfaz especifica y proporcione feedback.
- Comprender la estructura de un nodo cliente de Acciones en Python que pueda realizar peticiones.
- Integrar y aplicar los conceptos aprendidos sobre Acciones en ROS2 mediante el desarrollo de una aplicación práctica que controle el movimiento de una tortuga en el simulador turtlesim, utilizando Acciones para realizar un movimiento predeterminado y proporcione feedback acerca del mismo.

Índice general

1 Programación de Acciones (Actions)

En esta práctica, aprenderemos cómo crear un servidor de Actions y un cliente en ROS2 usando Python. Las *Actions* en ROS 2 proporcionan una forma de comunicación asíncrona entre nodos. Los clientes de acción envían solicitudes de objetivo a los servidores de acción, y estos, a su vez, envían retroalimentación y resultados a los clientes de Actions.

1.1 Definición de la Interfaz de Actions

Las acciones se especifican usando una forma del IDL (Lenguaje de Descripción de Interfaz) de Mensajes de ROS. La especificación contiene tres secciones, cada una de las cuales es una especificación de mensaje:

- **Objetivo (Goal):** Describe lo que la acción debe lograr y cómo debe hacerlo. Se envía al servidor de acción cuando se solicita ejecutar una acción.
- **Resultado (Result):** Describe el resultado de una acción. Se envía del servidor al cliente cuando la ejecución de la acción termina, ya sea con éxito o no.
- **Retroalimentación (Feedback):** Describe el progreso hacia la finalización de una acción. Se envía al cliente de la acción desde el servidor de acción entre el inicio de la ejecución de la acción y antes de que la acción se complete. Estos datos son utilizados por el cliente para comprender el progreso de la ejecución de la acción.

Cualquiera de estas secciones puede estar vacía. Entre cada una de las tres secciones hay una línea que contiene tres guiones, `---`. Las especificaciones de acción se almacenan en un archivo que termina en `.action`. Hay una especificación de acción por archivo `.action`.

```
Ejemplo:
# Definir un objetivo para mover el robot a una posición específica
float64 x  # Coordenada X destino
float64 y  # Coordenada Y destino
---
# Definir el resultado que se publicará después de que termine la ejecución de la acción.
bool success  # Verdadero si el robot alcanzó la posición, falso en caso contrario
float64 final_x  # Coordenada X final
float64 final_y  # Coordenada Y final
---
```

```
# Definir un mensaje de retroalimentación que se publicará durante la ejecución de la acción.
float64 current_x  # Coordenada X actual
float64 current_y  # Coordenada Y actual
float32 percent_complete  # Porcentaje de la trayectoria completada
```

En este ejemplo:

- La sección del objetivo define dos valores para especificar la posición objetivo a la que se debe mover el robot.
- La sección de retroalimentación proporciona información en tiempo real sobre la posición actual del robot y el porcentaje de la trayectoria que se ha completado hasta el momento
- La sección de resultado proporciona información sobre si el robot alcanzó con éxito la posición objetivo y, en tal caso, cuál es la posición final alcanzada.

1.2 Herramientas de monitorización por línea de comandos

Las acciones, como los topics y los servicios, pueden ser monitorizados desde la línea de comandos.

La herramienta de línea de comandos, `ros2 action`, será capaz de:

- (list) listar los nombres de acción asociados con cualquier servidor de acción o cliente de acción en ejecución.
- (list) listar servidores de acción y clientes de acción.
- (info) mostrar los objetivos activos en un servidor de acción.
- (info)mostrar los argumentos para un objetivo de acción.
- (info) mostrar el tipo de objetivo, retroalimentación y resultado de una acción.
- (send_goal) llamar a una acción, mostrar retroalimentación mientras se recibe, mostrar el resultado cuando se recibe y cancelar la acción (cuando la herramienta se termina prematuramente).

Cada acción será listada y tratada como una entidad única por esta herramienta. A continuación se muestra un ejemplo con la acción `/turtle1/rotate_absolute` (previamente lanzando el simulador). Comprobamos que esta la acción activa con este comando que además nos dice el tipo de interfaz que usa:

```
ros2 action list -t
```

Podemos obtener más información con:

```
ros2 action info /turtle1/rotate_absolute
```

```
ros2 interface show turtlesim/action/RotateAbsolute
```

Esto nos devolverá lo siguiente:

```
# The desired heading in radians

float32 theta

---

# The angular displacement in radians to the starting position

float32 delta

---

# The remaining rotation in radians

float32 remaining
```

Por último, para llamar a la acción podemos ejecutar:

```
ros2 action send_goal -f /turtle1/rotate_absolute
↪  turtlesim/action/RotateAbsolute "theta: 1.0"
```

El argumento de -f nos proporciona el feedback que en este caso es el error entre el ángulo deseado y el actual.

1.3 Estados del Objetivo

El servidor de acción mantiene una máquina de estados para cada objetivo que acepta de un cliente. Los objetivos rechazados no forman parte de la máquina de estados.

1.3.1 Máquina de Estados del Objetivo de actions

Tal y como se puede ver en la figura 1.1 hay tres estados activos:

- `ACCEPTED` - El objetivo ha sido aceptado y está esperando ejecución.
- `EXECUTING` - El objetivo está siendo ejecutado actualmente por el servidor de acción.

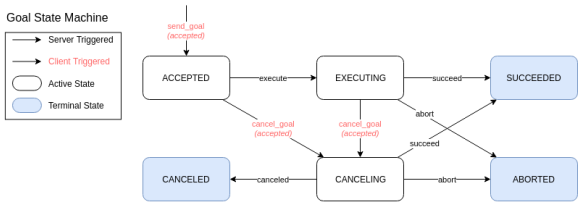

Figura 1.1: Máquina de estados de los objetivos de las acciones

- `CANCELING` - El cliente ha solicitado que el objetivo sea cancelado y el servidor de acción ha aceptado la solicitud de cancelación. Este estado es útil para cualquier "limpieza" definida por el usuario que el servidor de acción pueda tener que hacer.

Y tres estados terminales:

- `SUCCEEDED` - El objetivo fue alcanzado con éxito por el servidor de acción.
- `ABORTED` - El objetivo fue terminado por el servidor de acción sin una solicitud externa.
- `CANCELED` - El objetivo fue cancelado después de una solicitud externa de un cliente de acción.

Transiciones de estado desencadenadas por el servidor de acción según su comportamiento diseñado:

- `execute` - Comenzar la ejecución de un objetivo aceptado.
- `succeed` - Notificar que el objetivo se completó con éxito.
- `abort` - Notificar que se encontró un error durante el procesamiento del objetivo y tuvo que ser abortado.
- `canceled` - Notificar que la cancelación del objetivo se completó con éxito.

Transiciones de estado desencadenadas por el cliente de acción:

- `send_goal` - Un objetivo es enviado al servidor de acción. La máquina de estados solo se inicia si el servidor de acción acepta el objetivo.
- `cancel_goal` - Solicitar que el servidor de acción deje de procesar el objetivo. Una transición solo ocurre si el servidor de acción acepta la solicitud de cancelar el objetivo.

1.4 Escribir un Servidor de Acciones

1.4.1 Examinar el Código del Servidor de Acciones

Navega y abre el fichero `~/ros2_ws/src/ROS2_examples/rclpy/actions/minimal_action_server/examples_rclpy_minimal_action_server/server.py` con tu editor de texto favorito.

Vamos a comenzar con el servidor de la acción. Primero, se importa el tipo de acción `Fibonacci` del paquete `example_interfaces`. A continuación, se importa la biblioteca de cliente Python de ROS 2 y, específicamente, las clases y módulos necesarios.

```
from example_interfaces.action import Fibonacci
import rclpy
from rclpy.action import ActionServer, CancelResponse, GoalResponse
from rclpy.callback_groups import ReentrantCallbackGroup
from rclpy.executors import MultiThreadedExecutor
from rclpy.node import Node
```

La clase `MinimalActionServer` inicializa el nodo con el nombre `minimal_action_server`. Luego, crea un servidor de acción y define su tipo, nombre y callbacks.

```
def __init__(self):
    super().__init__('minimal_action_server')
    self._action_server = ActionServer(
        self,
        Fibonacci,
        'fibonacci',
        execute_callback=self.execute_callback,
        callback_group=ReentrantCallbackGroup(),
        goal_callback=self.goal_callback,
        cancel_callback=self.cancel_callback)
```

El parámetro `callback_group=ReentrantCallbackGroup()` en la creación del servidor de acción especifica el grupo de *callbacks* que será utilizado por el servidor de acción. En ROS 2, un grupo de *callbacks* es esencialmente una manera de agrupar diferentes *callbacks* para que se manejen de una manera particular por el *executor*.

Hay dos tipos de grupos de *callback* en ROS 2:

- `MutuallyExclusiveCallbackGroup`: Este grupo asegura que solo un *callback* en el grupo se ejecuta a la vez. Si hay múltiples *callbacks* en este grupo que están listos para ser ejecutados, se ejecutarán uno tras otro, no simultáneamente.

- `ReentrantCallbackGroup`: Este grupo permite que múltiples *callbacks* se ejecuten simultáneamente. Si hay múltiples *callbacks* en este grupo que están listos para ser ejecutados, pueden ejecutarse en paralelo si el *executor* tiene múltiples *threads*.

Se especifica un `ReentrantCallbackGroup`, lo que significa que los *callbacks* asociados con este servidor de acción (como `execute_callback`, `goal_callback`, y `cancel_callback`) pueden ser

ejecutados en paralelo. Esto puede ser útil si, por ejemplo, el servidor de acción está manejando múltiples *goals* simultáneamente y cada *goal* está en una fase diferente de ejecución.

Este diseño permite una mayor concurrencia y puede mejorar el rendimiento del servidor de acción, especialmente en situaciones donde el servidor de acción necesita manejar múltiples *requests* simultáneamente.

- **Goal Callback (`goal_callback`):** Este callback se invoca cuando se recibe una nueva solicitud de acción. Permite aceptar o rechazar nuevas solicitudes de acción. En el código proporcionado, simplemente se registra la recepción de una solicitud de acción y se acepta.

- **Cancel Callback (`cancel_callback`):** Este callback se invoca cuando se recibe una solicitud para cancelar una acción en curso. Permite manejar cómo se deben cancelar las acciones. En el código proporcionado, se registra la recepción de una solicitud de cancelación y se acepta.

- **Execute Callback (`execute_callback`):** Este callback se invoca para ejecutar la acción. También es responsable de proporcionar retroalimentación durante la ejecución de la acción. En el código proporcionado, se ejecuta una acción que calcula la secuencia de Fibonacci hasta un cierto número, proporcionando retroalimentación en cada paso.

```
def goal_callback(self, goal_request):
    self.get_logger().info('Received goal request')
    return GoalResponse.ACCEPT

def cancel_callback(self, goal_handle):
    self.get_logger().info('Received cancel request')
    return CancelResponse.ACCEPT

async def execute_callback(self, goal_handle):
    self.get_logger().info('Executing goal...')
    feedback_msg = Fibonacci.Feedback()
    feedback_msg.sequence = [0, 1]
    for i in range(1, goal_handle.request.order):
        if goal_handle.is_cancel_requested:
            goal_handle.canceled()
            self.get_logger().info('Goal canceled')
            return Fibonacci.Result()
        feedback_msg.sequence.append(feedback_msg.sequence[i] +
        ↪  feedback_msg.sequence[i-1])
        self.get_logger().info('Publishing feedback:
        ↪  {0}'.format(feedback_msg.sequence))
        goal_handle.publish_feedback(feedback_msg)
        time.sleep(1)
    goal_handle.succeed()
    result = Fibonacci.Result()
    result.sequence = feedback_msg.sequence
    self.get_logger().info('Returning result: {0}'.format(result.sequence))
    return result
```

Finalmente, el método `main` inicia los servicios de ROS, instancia la clase `MinimalActionServer` para crear el nodo del servidor de acción y mantiene el nodo en ejecución para manejar los callbacks utilizando un `MultiThreadedExecutor`.

```
def main(args=None):
    rclpy.init(args=args)

    minimal_action_server = MinimalActionServer()

    executor = MultiThreadedExecutor()

    rclpy.spin(minimal_action_server, executor=executor)

    minimal_action_server.destroy()
    rclpy.shutdown()


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

`MultiThreadedExecutor` es una clase diseñada para permitir la ejecución concurrente de múltiples tareas, como la manipulación de callbacks, en paralelo. Este executor crea varios threads, cada uno de los cuales puede procesar callbacks de los nodos de ROS. La ejecución multithreaded permite que múltiples callbacks se ejecuten simultáneamente en diferentes threads, lo cual es crucial para garantizar que el servidor de acción pueda manejar múltiples requests simultáneamente sin bloqueo, especialmente en escenarios donde las operaciones pueden llevar algún tiempo en completarse.

En el código proporcionado, el `MultiThreadedExecutor` se instancia y luego se pasa como argumento a la función `rclpy.spin()`, junto con la instancia de `MinimalActionServer`. La función `rclpy.spin()` mantiene el nodo `minimal_action_server` en ejecución y utiliza el `executor` proporcionado para procesar los callbacks de los nodos. Este diseño permite que `MinimalActionServer` maneje múltiples goals en paralelo, cada uno en su propio thread, mejorando así la eficiencia y la capacidad de respuesta del servidor de acción.

El uso de `MultiThreadedExecutor` es particularmente beneficioso en entornos donde las operaciones pueden ser bloqueantes o llevar mucho tiempo, ya que permite que el servidor de acción continúe procesando otros requests mientras espera que las operaciones en curso finalicen. Esto mejora la eficiencia y el rendimiento del servidor de acción, permitiendo una mayor concurrencia y una mejor utilización de los recursos del sistema.

1.4.2 Prueba del Servidor de Acciones desde la Línea de Comando

Para probar el servidor de acciones, primero necesitaremos asegurarnos de que el servidor esté en ejecución. Podemos hacerlo iniciando el servidor desde la línea de comando ejecutando el siguiente comando:

```
ros2 run examples_rclpy_minimal_action_server server
```

Con el servidor en ejecución, ahora podemos interactuar con él utilizando herramientas de línea de comando proporcionadas por ROS 2. La herramienta `ros2 action` proporciona varias subcomandos para interactuar con servidores de acción.

Listar Acciones: Para listar los servidores de acción disponibles, ejecuta el siguiente comando:

```
ros2 action list -t
```

Deberías ver la acción `/fibonacci` en la lista.

Mostrar Información de la Acción: Para obtener información sobre una acción específica, usa el subcomando `info`:

```
ros2 action info /fibonacci
```

Esto mostrará información sobre la acción `/fibonacci`, incluyendo los tipos de mensajes utilizados para los *goals*, *results* y *feedback*.

Enviar un Goal: Podemos enviar un *goal* al servidor de acción utilizando el subcomando `send_goal`:

```
ros2 action send_goal /fibonacci example_interfaces/action/Fibonacci "{order:
↪  10}"
```

Esto enviará un *goal* al servidor para calcular la secuencia de Fibonacci hasta el número 10. También puedes observar el feedback en tiempo real en la línea de comando mientras el servidor procesa el *goal*.

Cancelar un Goal: La cancelación de un *goal* es una operación que puede realizarse en un servidor de acción si, por alguna razón, decides que ya no deseas continuar con la ejecución de un *goal* específico. Esto puede ser útil, por ejemplo, si un *goal* está tomando demasiado tiempo y ya no es necesario, o si se ha producido una condición que hace que el *goal* ya no sea deseable.

Para cancelar un *goal* en progreso, ROS 2 proporciona el subcomando `cancel` de la herramienta `ros2 action`. Aquí está el comando que necesitarás utilizar para cancelar un *goal*:

```
ros2 action cancel_goal /fibonacci <goal_id>
```

En este comando:

- `fibonacci` es el nombre de la acción donde se está ejecutando el *goal* que deseas cancelar.
- `goal_id` es un marcador de posición para el ID del *goal* que deseas cancelar. Cada *goal* enviado a un servidor de acción recibe un ID único que se puede utilizar para referirse a él en operaciones futuras, como la cancelación.

Deberás reemplazar `goal_id` con el ID real del *goal* que deseas cancelar. Puedes obtener el ID del *goal* a partir del *feedback* o la información de estado proporcionada por el servidor de acción.

Una vez que se emite el comando de cancelación, el servidor de acción recibirá una solicitud de cancelación para el *goal* especificado. Si el *goal* se cancela con éxito, el servidor de acción proporcionará una notificación de que el *goal* ha sido cancelado.

1.5 Escribir un Cliente de Acciones

1.5.1 Examinar el Código del Cliente de Acciones

Navega y abre el fichero correspondiente con tu editor de texto favorito.

Vamos a comenzar con el cliente de la acción. Primero, se importan los tipos de mensajes y acciones necesarios, así como las clases y módulos necesarios de ROS 2.

```
from action_msgs.msg import GoalStatus
from example_interfaces.action import Fibonacci

import rclpy
from rclpy.action import ActionClient
from rclpy.node import Node
```

La clase `MinimalActionClient` inicializa el nodo con el nombre `minimal_action_client`. Luego, crea un cliente de acción que se conecta al servidor de acción `fibonacci`.

```
class MinimalActionClient(Node):

def __init__(self):
    super().__init__('minimal_action_client')
    self._action_client = ActionClient(self, Fibonacci, 'fibonacci')
```

A continuación, se definen varios *callbacks* para manejar diferentes etapas del ciclo de vida de un *goal*:

- **Goal Response Callback (`goal_response_callback`):** Este *callback* se invoca cuando se recibe una respuesta a una solicitud de *goal*. Verifica si el *goal* fue aceptado y, en caso afirmativo, inicia la solicitud para obtener el resultado.
- **Feedback Callback (`feedback_callback`):** Este *callback* se invoca cada vez que se recibe retroalimentación del servidor de acción. En este caso, simplemente imprime la secuencia de Fibonacci hasta el momento.
- **Get Result Callback (`get_result_callback`):** Este *callback* se invoca cuando se recibe el resultado del *goal*. Imprime el resultado o un mensaje de error si el *goal* falló.

```
def goal_response_callback(self, future):
    goal_handle = future.result()
    if not goal_handle.accepted:
    self.get_logger().info('Goal rejected :(')
    return

    self.get_logger().info('Goal accepted :)')

    self._get_result_future = goal_handle.get_result_async()
    self._get_result_future.add_done_callback(self.get_result_callback)

def feedback_callback(self, feedback):
    self.get_logger().info('Received feedback:
    ↪  {0}'.format(feedback.feedback.sequence))

def get_result_callback(self, future):
    result = future.result().result
    status = future.result().status
    if status == GoalStatus.STATUS_SUCCEEDED:
        self.get_logger().info('Goal succeeded! Result:
        ↪  {0}'.format(result.sequence))
    else:
        self.get_logger().info('Goal failed with status: {0}'.format(status))

    # Shutdown after receiving a result
    rclpy.shutdown()
```

El método `send_goal` es el encargado de enviar el *goal* al servidor de acción. Primero, espera a que el servidor de acción esté disponible. Luego, crea un mensaje de *goal*, establece el orden de la secuencia de Fibonacci, y envía el *goal* al servidor de acción.

```
def send_goal(self):
    self.get_logger().info('Waiting for action server...')
    self._action_client.wait_for_server()
    goal_msg = Fibonacci.Goal()
    goal_msg.order = 10

    self.get_logger().info('Sending goal request...')

    self._send_goal_future = self._action_client.send_goal_async(
        goal_msg,
        feedback_callback=self.feedback_callback)

    self._send_goal_future.add_done_callback(self.goal_response_callback)
```

Finalmente, el método `main` inicia los servicios de ROS, instancia la clase `MinimalActionClient` para crear el nodo del cliente de acción, envía un *goal* y mantiene el nodo en ejecución para manejar los *callbacks*.

```
def main(args=None):
    rclpy.init(args=args)

    action_client = MinimalActionClient()

    action_client.send_goal()

    rclpy.spin(action_client)

    # Shutdown ROS
    rclpy.shutdown()
if name == 'main':
main()
```

1.5.2 Prueba del Cliente de Acciones desde la Línea de Comando

Para probar el cliente de acciones que hemos creado para la secuencia de Fibonacci, necesitaremos seguir los siguientes pasos:

1. Primero, asegúrate de que el servidor de acciones esté en ejecución. Si no está en ejecución, inícialo con lo que vimos anteriormente.

2. Ahora, en otra terminal, ejecuta el cliente de acciones con el siguiente comando:

   ```
   ros2 run examples_rclpy_minimal_action_client client N
   ```

   Donde N es el argumento de entrada con la orden de Fibonacci.

3. Una vez que el cliente esté en ejecución, deberías ver mensajes en la terminal indicando que se ha enviado una solicitud de objetivo al servidor, y luego deberías ver mensajes de retroalimentación a medida que el servidor procesa el objetivo.

4. El cliente imprimirá la secuencia de Fibonacci en la terminal a medida que reciba la retroalimentación del servidor. También imprimirá un mensaje una vez que se complete el objetivo, indicando si el objetivo se logró con éxito o si falló.

5. También puedes monitorear la comunicación entre el cliente y el servidor utilizando herramientas de línea de comandos de ROS 2. Por ejemplo, para listar los servidores de acción y clientes de acción en ejecución, puedes usar el siguiente comando:

   ```
   ros2 action list
   ```

6. Para obtener información detallada sobre la acción de Fibonacci, puedes usar el siguiente comando:

   ```
   ros2 action info /fibonacci
   ```

7. Si deseas obtener más información sobre los objetivos activos en un servidor de acción, puedes usar el siguiente comando:

```
ros2 action list -t /fibonacci
```

Al seguir estos pasos, podrás observar cómo el cliente de acciones interactúa con el servidor de acciones para calcular una secuencia de Fibonacci, y cómo se puede monitorear y verificar esta comunicación desde la línea de comandos. Esto te proporcionará una visión clara de cómo funcionan las Acciones en ROS 2 y cómo se pueden utilizar para facilitar la comunicación asíncrona entre nodos en un sistema robótico.

2 Ejercicio:Acción para Dibujar un Círculo con Turtlesim

2.1 Ejercicio: Servidor de Acciones

En este ejercicio, se pide crear un servidor de acciones en ROS 2 que controle una tortuga en el simulador `turtlesim` para que dibuje un círculo de radio X. Para ello utilice la misma interfaz que `/turtle1/rotate_absolute`.

La acción debe tomar como meta el radio X del círculo que se desea dibujar. Durante la ejecución de la acción, el servidor debe proporcionar feedback sobre los radianes que la tortuga ha dibujado del círculo. Al completar el círculo, el servidor debe retornar el total de radianes dibujados como resultado (que debe ser 2π si el círculo se completa exitosamente).

Especificaciones:

- El fichero se llamará `circle_action_server.py`.
- La acción se llamará `draw_circle`
- Utilice la interfaz de acción `/turtle1/rotate_absolute` para controlar la orientación de la tortuga.
- La velocidad lineal v y la velocidad angular ω de la tortuga deben ser controladas de tal manera que se dibuje un círculo perfecto. Se puede fijar una velocidad lineal de 1.
- El feedback debe ser proporcionado en radianes, indicando cuánto del círculo ha sido dibujado por la tortuga.
- El resultado final debe ser los radianes totales dibujados por la tortuga.

Ecuaciones:

1. **Relación entre Velocidad Lineal y Angular para un Círculo:**

La relación entre la velocidad lineal v y la velocidad angular ω para un círculo es dada por la ecuación:

$$v = \omega \cdot X$$

En ROS 2, la velocidad lineal y la velocidad angular se expresan a través de un mensaje de tipo `geometry_msgs/Twist`. La componente `linear.x` representa la velocidad lineal v y la componente `angular.z` representa la velocidad angular ω. Entonces, la ecuación anterior se puede expresar en términos de estos componentes como sigue:

$$\texttt{linear.x} = \texttt{angular.z} \cdot X$$

Donde:

- `linear.x` es la velocidad lineal,
- `angular.z` es la velocidad angular,
- X es el radio del círculo.

2. **Cálculo de los Radianes Dibujados:**

La cantidad de radianes θ dibujados en un círculo es dada por la ecuación:

$$\theta = t \cdot \texttt{angular.z}$$

Donde:

- t es el tiempo transcurrido desde que comenzó a dibujar el círculo,
- `angular.z` es la velocidad angular.

Pseudocódigo:

A continuación se muestra en pseudocódigo lo que habría que hacer para este servidor de acciones:

1. Importar módulos necesarios
2. Definir la clase `CircleServer` que hereda de `Node`
    (a) En el constructor:

i. Llamar al constructor de la clase padre con el nombre 'circle_action_server'
ii. Iniciar el servidor de acciones para 'draw_circle'
iii. Crear un publicador para 'turtle1/cmd_vel'
iv. Inicializar la variable de tiempo de inicio a `None`
v. Crear temporizador para periodo de muestreo en bucle (ej. 10 Hz)

(b) Definir `execute_callback`:

i. Mostrar un mensaje indicando la ejecución del objetivo
ii. Obtener el radio deseado del objetivo
iii. Calcular las velocidades lineal y angular
iv. Crear y publicar un mensaje `Twist`
v. Registrar el tiempo de inicio
vi. Calcular el tiempo total necesario para completar el círculo
vii. Hacer un bucle infinito mientras el programa esté funcionando:
   A. Publicar el mensaje `Twist`
   B. Calcular el tiempo transcurrido desde el inicio
   C. Si se ha alcanzado el tiempo total, detener el bucle
   D. Calcular y mostrar los radianes dibujados
   E. Enviar retroalimentación
   F. Dormir durante un corto periodo
viii. Detener el movimiento de la tortuga
ix. Enviar resultado

3. Definir la función `main`:

(a) Iniciar `rclpy`

(b) Crear una instancia de `CircleServer`

(c) Ejecutar el servidor

(d) Destruir el nodo y cerrar `rclpy`

4. Si el archivo se ejecuta como programa principal, llamar a la función `main`

Aquí se muestran algunas notas que te pueden ser útiles:

```
# Condición del bucle que es true mientras el nodo esté vivo
rclpy.ok()

#Sacar el tiempo actual del reloj
self.get_clock().now()

#Calcular el tiempo que ha pasado en segundos:
duration = self.get_clock().now() - self.start_time
elapsed_time = duration.nanoseconds / 1e9
```

```
    #  Crear temporizador para periodo de muestreo de 10hz
    self.rate = self.create_rate(10)  # 10Hz for a 0.1 second sleep interval

    # Dormir en el bucle
    self.rate.sleep()
```

2.2 Ejercicio: Cliente de Acciones

Ahora haz un cliente que ejecute una acción de dibujar con radio 1 y cuando termine una con radio 2. El resultado tiene que ser similar al mostrado en la figura 2.1.

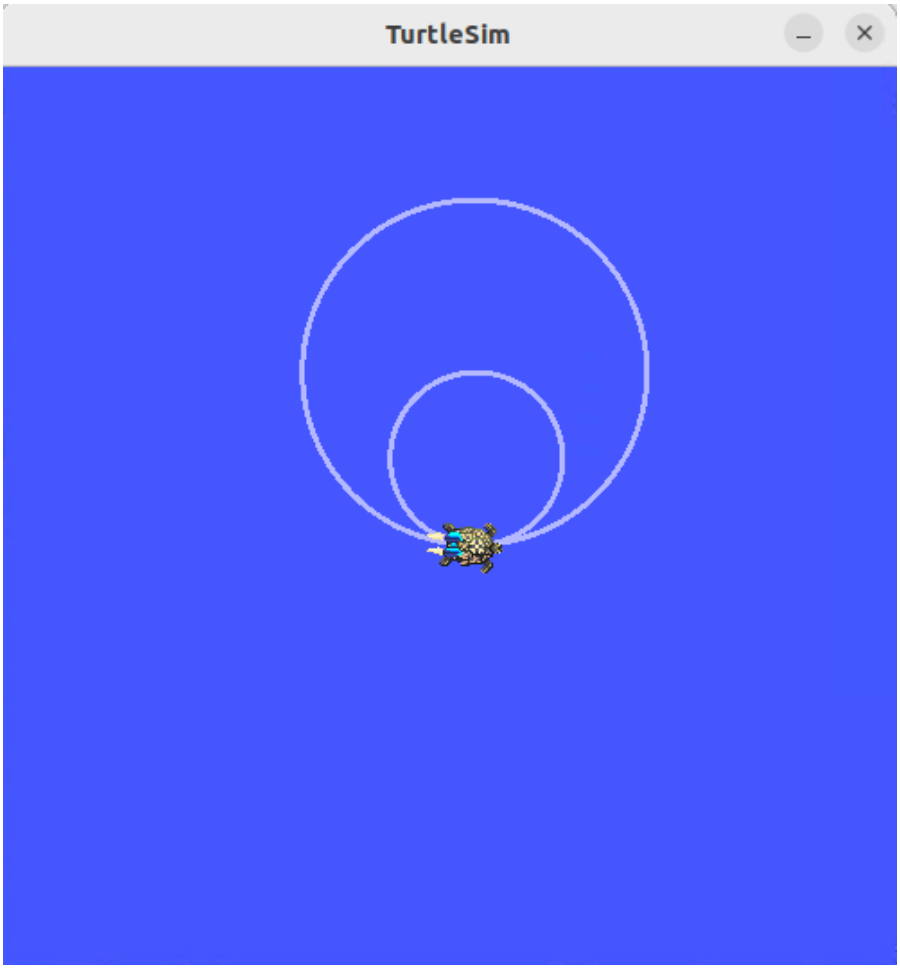

Figura 2.1: Resultado

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.