UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escola Tècnica Superior d'Enginyeria Informàtica

etsinf

Tortuga Exploradora Ros2 2025 -2026

Aoiz Canillas, Kai

Li, ZhiHui

Profesores: Patricia Balbastre Betoret,

Juan Francisco Blanes Noguera

Indice

1. Introducción

En este trabajo se desarrolla una aplicación en ROS 2 (Humble) sobre el simulador turtlesim, cuyo propósito principal es practicar el diseño de sistemas distribuidos mediante topics, servicios y acciones. Para ello se implementa un escenario con dos tortugas:

- **turtle1:** tortuga principal del simulador, que se puede mover manualmente (por ejemplo, con turtle_teleop_key) o dejar en reposo.

- **explorer:** una segunda tortuga creada dinámicamente por el sistema (spawn) cuya misión es seguir a turtle1 y proporcionar información del estado del sistema.

La solución se implementa en dos variantes que comparten la misma lógica de seguimiento:

1) **Versión Service:** la información se expone mediante un servicio turtle_info y un cliente la consulta de forma periódica.

2) **Versión Action:** la información se expone mediante una acción turtle_info que envía feedback y finaliza cuando explorer alcanza a turtle1.

2. Descripción de la implementación

2.1. Estructura del repositorio / paquetes

La solución se divide en dos paquetes ROS 2:

- **turtle_tracker_interfaces**

Contiene las definiciones de interfaces (.srv y .action) para el servicio/acción de información.

- **turtle_tracker**

Contiene los nodos en Python que implementan el comportamiento del sistema (control, servidor/cliente de service o action) y los archivos launch para ejecutar el sistema completo.

````
Fragmento de dependencias en turtle_tracker (package.xml):

```xml
<depend>rclpy</depend>
<depend>turtlesim</depend>
<depend>geometry_msgs</depend>
<depend>turtle_tracker_interfaces</depend>
```
````

2.2. Flujo general (Service)

En el modo Service, el sistema se compone de:

- **explorer_service_node.py**

  Spawnea explorer, realiza el seguimiento mediante control proporcional y ofrece el servicio turtle_info devolviendo información actual.

- **service_client_node.py**

  Actúa como “monitor” automático. Cada segundo invoca el servicio turtle_info e imprime la información de forma estructurada y legible.

- **service_system.launch.xml**

  Lanza turtlesim y ambos nodos anteriores en un único comando, además de permitir parametrizar la posición inicial de explorer.

2.3. Flujo general (Action)

En el modo Action, el sistema es similar pero cambia la interfaz:

- **explorer_action_node.py**

  Mantiene spawn y seguimiento, y además implementa un Action Server turtle_info con feedback y resultado final.

- **action_client_node.py**

  Implementa un Action Client que monitoriza la distancia entre tortugas y activa la acción cuando detecta movimiento según un umbral. Muestra feedback y, al finalizar, el resultado.

- **action_system.launch.xml**

  Lanza turtlesim y los nodos del modo Action con parámetros de spawn.

3. Interfaces ROS2 implementadas

3.1. Topics

Suscripciones:

- /turtle1/pose (turtlesim/msg/Pose)

- /explorer/pose (turtlesim/msg/Pose)

Publicación:

- /explorer/cmd_vel (geometry_msgs/msg/Twist)

Los topics de pose proporcionan el estado (posición, orientación y velocidades) publicado por el simulador, mientras que cmd_vel representa la orden de control enviada a la tortuga explorer.

Ejemplo de suscripciones y publicador (explorer_service_node.py):

```python
self.create_subscription(Pose, '/turtle1/pose', self.turtle1_callback,
10, callback_group=self.callback_group)
self.create_subscription(Pose, '/explorer/pose',
self.explorer_callback, 10, callback_group=self.callback_group)
self.cmd_pub = self.create_publisher(Twist, '/explorer/cmd_vel', 10)
```

3.2. Servicio `turtle_info` (E3 + E4)

Nombre del servicio: **turtle_info**

Tipo: turtle_tracker_interfaces/srv/TurtleInfo

El servicio se define con request vacío y response con:

- Pose de turtle1 (x, y, theta)
- Pose de explorer (x, y, theta)
- Velocidades de turtle1 (del mensaje Pose)
- Velocidad comandada a explorer (del último Twist publicado)
- Distancia euclídea entre ambas tortugas

Este enfoque simplifica el cliente, ya que la consulta es únicamente lectura del estado actual.

````
Definición (TurtleInfo.srv):
```text
# Request (vacía)
float32 turtle1_x
float32 turtle1_y
float32 turtle1_theta
float32 explorer_x
float32 explorer_y
float32 explorer_theta
float32 explorer_linear_velocity
float32 explorer_angular_velocity
float32 turtle1_linear_velocity
float32 turtle1_angular_velocity
float32 distance
````

3.3. Acción turtle_info (E6)

Nombre de la acción: **turtle_info**

Tipo: turtle_tracker_interfaces/action/TurtleInfo
````

La acción tiene goal vacío y devuelve un result final con la misma información del servicio. Además, publica feedback periódico, permitiendo observar la evolución del seguimiento. La acción finaliza cuando se cumple la condición de éxito (distancia por debajo de un umbral), modelando una tarea con inicio, progreso y final

````
Definición (TurtleInfo.action):

```text
# Goal (vacío)
---

# Result (información final)
float32 turtle1_x
...
float32 distance

---

# Feedback (información periódica)
float32 turtle1_x
...
float32 distance

---
````
````

4. Pruebas y resultados

4.1. Compilación

Para la compulación de nuestro trabajo haremos el colcon build en nuestra carpeta raíz que en este caso es ros2_eoii

```
5.  ```bash
6.  colcon build
7.  source install/setup.bash
```

Resultado esperado:

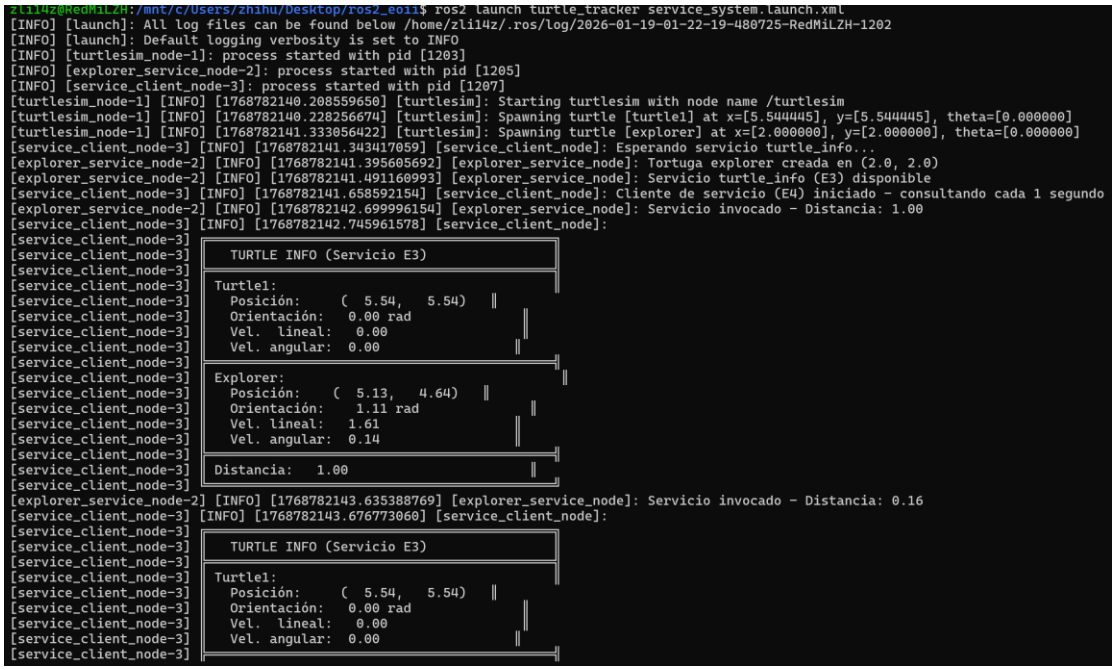
```
zli14z@RedMiLZH:/mnt/c/Users/zhihu/Desktop/ros2_eoii$ colcon build
Starting >>> turtle_tracker_interfaces
[Processing: turtle_tracker_interfaces]
Finished <<< turtle_tracker_interfaces [58.4s]
Starting >>> turtle_tracker
Finished <<< turtle_tracker [10.9s]

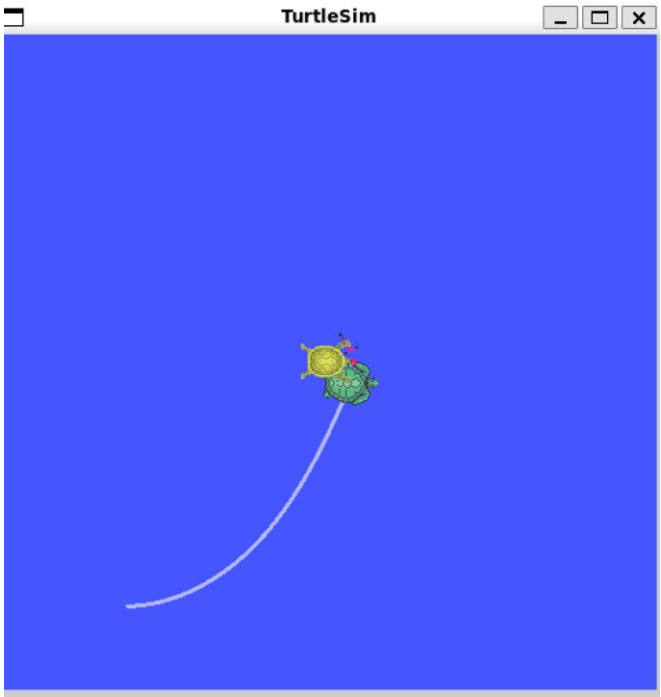
Summary: 2 packages finished [1min 10s]
zli14z@RedMiLZH:/mnt/c/Users/zhihu/Desktop/ros2_eoii$ source install/setup.bash
```

4.2. Prueba del sistema Service

```
ros2 launch turtle_tracker service_system.launch.xml
```

Este comando nos muestra lo siguiente en el terminal:





En la terminal se nos va estar mostrando cada segundo las posiciones de las dos tortugas.

Ahora vamos a probar a llamar el launch pero le pasaremos por consola los parámetros de la posición inicial de la tortuga explorer en este caso hemos eleigod x=5 y y=5:

```
ros2 launch turtle_tracker service_system.launch.xml spawn_x:=5.0
spawn_y:=5.0
```

Se nos mostrara por terminal lo mismo que cuando lo hacemos sin pasar parámetros pero en el simulador podemos ver que la tortuga explorer ha aperecido mas cerca que antes.

4.3. Prueba del sistema Action

```
ros2 launch turtle_tracker action_system.launch.xml
```

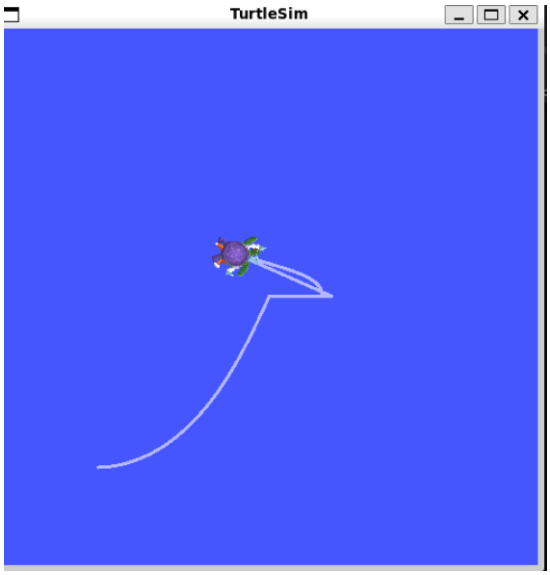
Este comando nos muestra lo siguiente en pantalla:

```
[turtlesim_node-1] [INFO] [1768783537.498025347] [turtlesim]: Starting turtlesim with node name /turtlesim
[turtlesim_node-1] [INFO] [1768783537.516389486] [turtlesim]: Spawning turtle [turtle1] at x=[5.544445], y=[5.544445], theta=[0.000000]
[turtlesim_node-1] [INFO] [1768783540.117311863] [turtlesim]: Spawning turtle [explorer] at x=[2.000000], y=[2.000000], theta=[0.000000]
[explorer_action_node-2] [INFO] [1768783540.288094513] [explorer_action_node]: Tortuga explorer creada en (2.0, 2.0)
[explorer_action_node-2] [INFO] [1768783540.343892838] [explorer_action_node]: Action Server turtle_info (E6) disponible
[action_client_node-3] [INFO] [1768783540.364136496] [action_client_node]: Cliente de Action (E6) iniciado.  Esperando movimiento para activar.. .
[action_client_node-3] [INFO] [1768783540.747801381] [action_client_node]: Movimiento detectado (Distancia: 3.21). Activando Action (E6)...
[action_client_node-3] [INFO] [1768783540.777111162] [action_client_node]: Goal aceptada por Action Server (E6).
[explorer_action_node-2] [INFO] [1768783540.810952757] [explorer_action_node]: Action turtle_info aceptada (E6)
[action_client_node-3] [INFO] [1768783540.855141839] [action_client_node]:
[action_client_node-3]
[action_client_node-3]   TURTLE INFO (Action E6 - Feedback)
[action_client_node-3]
[action_client_node-3]  Turtle1:   (  5.54,   5.54)
[action_client_node-3]  Explorer:  (  4.20,   3.21)
[action_client_node-3]  Distancia:   2.70
[action_client_node-3]
[action_client_node-3] [INFO] [1768783541.866746680] [action_client_node]:
[action_client_node-3]
[action_client_node-3]   TURTLE INFO (Action E6 - Feedback)
[action_client_node-3]
[action_client_node-3]  Turtle1:   (  5.54,   5.54)
[action_client_node-3]  Explorer:  (  5.33,   5.09)
[action_client_node-3]  Distancia:   0.50
[action_client_node-3]
[action_client_node-3] [INFO] [1768783542.848232996] [action_client_node]:
[action_client_node-3]
[action_client_node-3]   TURTLE INFO (Action E6 - Feedback)
[action_client_node-3]
[action_client_node-3]  Turtle1:   (  5.54,   5.54)
[action_client_node-3]  Explorer:  (  5.50,   5.46)
[action_client_node-3]  Distancia:   0.09
[action_client_node-3]
[explorer_action_node-2] [INFO] [1768783542.859635942] [explorer_action_node]: Explorer ha alcanzado turtle1. Action finalizada (E6).
[action_client_node-3] [INFO] [1768783542.913307915] [action_client_node]:
[action_client_node-3]
[action_client_node-3]   ACTION COMPLETADA (E6)
[action_client_node-3]
[action_client_node-3]  Explorer alcanzó a Turtle1
[action_client_node-3]  Distancia final:   0.09
[action_client_node-3]
```

Como podemos ver el explorer al acercarse a la tortuga1 la acción finaliza, como se nos informa en la terminal.

Ahora al mover la tortuga1 a una distancia mayor que la distancia umbral que hayamos configurado (este caso 0.55) esta volverá a solicitar la acción, como se puede ver en la terminal:

```
[action_client_node-3] [INFO] [1768783828.414342770] [action_client_node]:
[action_client_node-3] ╔═══════════════════════════════════════╗
[action_client_node-3]   ACTION COMPLETADA (E6)
[action_client_node-3] ╠═══◆◆◆═══════════════════════════════╣
[action_client_node-3] ║ Explorer alcanzó a Turtle1            ║
[action_client_node-3] ║ Distancia final:   0.38               ║
[action_client_node-3] ╚═══════════════════════════════════════╝
[action_client_node-3] [INFO] [1768783828.756247821] [action_client_node]: Movimiento detectado (Distancia: 0.98). Activando Action (E6)...
[action_client_node-3] [INFO] [1768783828.783052406] [action_client_node]: Goal aceptada por Action Server (E6).
[explorer_action_node-2] [INFO] [1768783828.785279507] [explorer_action_node]: Action turtle_info aceptada (E6)
[action_client_node-3] [INFO] [1768783828.832540412] [action_client_node]:
[action_client_node-3] ╔═══════════════════════════════════════╗
[action_client_node-3]   TURTLE INFO (Action E6 - Feedback)
[action_client_node-3] ╠═══════════════════════════════════════╣
[action_client_node-3] ║ Turtle1:   (  5.39,   6.19)           ║
[action_client_node-3] ║ Explorer:  (  6.38,   5.91)           ║
[action_client_node-3] ║ Distancia:   1.03                     ║
[action_client_node-3] ╚═══════════════════════════════════════╝
[explorer_action_node-2] [INFO] [1768783829.813678454] [explorer_action_node]: Explorer ha alcanzado turtle1. Action finalizada (E6).
[action_client_node-3] [INFO] [1768783829.815519305] [action_client_node]:
[action_client_node-3] ╔═══════════════════════════════════════╗
[action_client_node-3]   TURTLE INFO (Action E6 - Feedback)
[action_client_node-3] ╠═══════════════════════════════════════╣
[action_client_node-3] ║ Turtle1:   (  4.87,   6.42)           ║
[action_client_node-3] ║ Explorer:  (  5.20,   6.32)           ║
[action_client_node-3] ║ Distancia:   0.35                     ║
[action_client_node-3] ╚═══════════════════════════════════════╝
[action_client_node-3] [INFO] [1768783829.867294500] [action_client_node]:
[action_client_node-3] ╔═══════════════════════════════════════╗
[action_client_node-3]   ACTION COMPLETADA (E6)
[action_client_node-3] ╠═══◆◆◆═══════════════════════════════╣
[action_client_node-3] ║ Explorer alcanzó a Turtle1            ║
[action_client_node-3] ║ Distancia final:   0.35               ║
[action_client_node-3] ╚═══════════════════════════════════════╝
```



Este sería el comando para pasar por terminal la posición incial de la tortuga explorer:

```
ros2 launch turtle_tracker action_system.launch.xml spawn_x:=5.0
spawn_y:=5.0
```

5. Problemas encontrados y soluciones

5.1. Sincronización con turtlesim: el servicio /spawn no está disponible al inicio

Descripción del problema:

Para crear la tortuga explorer se utiliza el servicio /spawn de turtlesim. Cuando se ejecuta el sistema con launch files, los nodos se arrancan de forma casi simultánea. En ese contexto, puede ocurrir que turtlesim todavía no haya terminado de inicializarse y, por tanto, el servicio /spawn aún no exista en el grafo de ROS 2. Si el nodo explorer intentara llamar a /spawn inmediatamente, la creación de la tortuga podría fallar.

Solución aplicada:

Se implementa una espera activa utilizando wait_for_service en bucle. De esta forma, el nodo no continúa hasta que el servicio esté realmente disponible.

Fragmento de código (explorer_service_node.py y explorer_action_node.py):

```
self.spawn_client = self.create_client(Spawn, '/spawn')
while not self.spawn_client.wait_for_service(timeout_sec=1.0):
    self.get_logger().info('Esperando /spawn...')
```

Justificación de la solución:

wait_for_service comprueba periódicamente la existencia del servidor. Cuando turtlesim termina de arrancar, el servicio aparece y el bucle finaliza, garantizando que la petición de spawn se realice en un momento válido.

5.2. Inicialización de suscripciones: poses no disponibles (variables en None)

Descripción del problema:

Durante los primeros instantes tras arrancar el sistema, las suscripciones a /turtle1/pose y /explorer/pose pueden tardar en recibir los primeros mensajes. En ese intervalo, las variables que almacenan las poses permanecen en None. Si el control proporcional o el servidor de servicio/acción intentan acceder a esos datos antes de que existan, se producirían cálculos incorrectos o errores por acceso a campos inexistentes.

Solución aplicada:

Se añaden comprobaciones de seguridad antes de ejecutar cálculos. En el control loop se retorna inmediatamente si las poses aún no han sido recibidas. En el servidor del servicio se devuelve una respuesta vacía y se registra una advertencia. En la acción, se espera brevemente y se reintenta.

Fragmentos de código:

````
Control loop (explorer_service_node.py y explorer_action_node.py):
```python
if self.turtle1_pose is None or self.explorer_pose is None:
    return
```
Servicio (explorer_service_node.py):
```python
if self.turtle1_pose is None or self.explorer_pose is None:
    self.get_logger().warning('Poses no disponibles aún')
    return response
```
Acción (explorer_action_node.py):
```python
if self.turtle1_pose is None or self.explorer_pose is None:
    time.sleep(0.5)
    continue
```
````

Justificación de la solución:

Estas condiciones evitan que el sistema realice cálculos con datos inexistentes durante el arranque y permiten que el sistema se estabilice automáticamente cuando comienzan a llegar mensajes por los topics.

5.3. Concurrencia y posibles bloqueos: múltiples callbacks (timer + suscripciones + service/action)

Descripción del problema:

Los nodos explorer realizan varias tareas de forma concurrente: procesan callbacks de suscripción (poses), ejecutan un timer de control a 10 Hz y además atienden peticiones de servicio o ejecutan un Action Server. En particular, el Action Server implementa un bucle de ejecución con sleeps y publicación de feedback. Si todo esto se ejecutara en un único hilo o con exclusión rígida entre callbacks, podría aparecer latencia, pérdida de fluidez en el control o incluso bloqueo temporal de algunas tareas.

Solución aplicada:

Se combinan dos mecanismos de ROS 2:

1) ReentrantCallbackGroup: permite que callbacks se ejecuten de forma reentrante en el mismo nodo.

2) MultiThreadedExecutor: permite ejecutar callbacks en varios hilos, evitando que un callback “largo” bloquee el resto.

Fragmentos de código:

Definición de callback group reentrante:

```python
self.callback_group = ReentrantCallbackGroup()
```

```

Asignación del callback group al timer y a las suscripciones:
```python
self.create_timer(0.1, self.control_loop,
callback_group=self.callback_group)
self.create_subscription(..., callback_group=self.callback_group)
```

Uso de ejecutor multihilo en main():
```python
executor = MultiThreadedExecutor()
executor.add_node(node)
executor.spin()
```
Justificación de la solución:

Con ReentrantCallbackGroup y MultiThreadedExecutor, el nodo puede seguir ejecutando el control y procesando suscripciones mientras atiende una petición de servicio o mientras el Action Server publica feedback, mejorando robustez y fluidez del seguimiento.

5.4. Parámetros de spawn fuera de rango para el mapa de TurtleSim

Descripción del problema:

TurtleSim opera en un mapa limitado (aproximadamente entre 0 y 11 en ambos ejes). Un valor incorrecto de spawn_x o spawn_y pasado por parámetros (por ejemplo desde launch) podría provocar una condición inicial inválida o no deseada, complicando las pruebas y pudiendo generar comportamientos extraños.

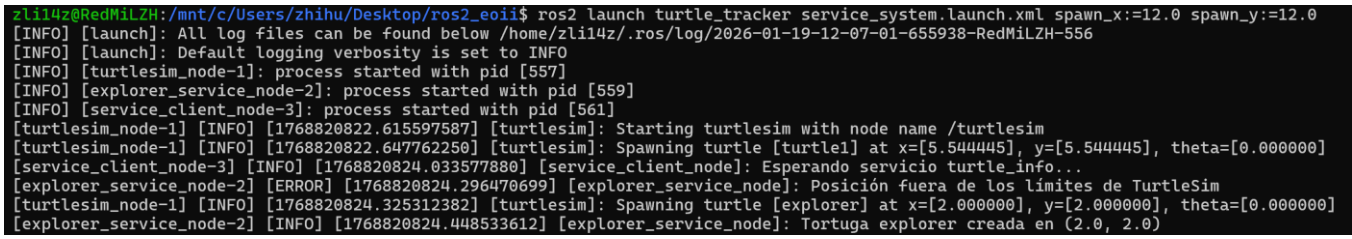
Solución aplicada:
```

Se valida el rango de los parámetros y, si se detectan valores fuera del intervalo, se corrigen a valores por defecto seguros.

Fragmento de código:

```
if not (0.0 <= x <= 11.0 and 0.0 <= y <= 11.0):
    self.get_logger().error('Posición fuera de los límites de
TurtleSim')
    x, y = 2.0, 2.0
```

Demostración en terminal:



Hemos creado la tortuga fuera de los límites del TurtleSim y por esta razón nos hace un feedback avisándonos que la tortuga se ha creado en los puntos x=2 y=2 que es la que hemos determinado en nuestro código.

Justificación de la solución:

La validación evita estados iniciales inválidos, aumenta la reproducibilidad del sistema y previene errores provocados por parámetros mal introducidos.

5.5. Gestión de estado en el Action Client: evitar envío repetido de goals

Descripción del problema:

El Action Client evalúa condiciones en un timer periódico. Sin control de estado, podría enviar múltiples goals consecutivos, generando ejecuciones simultáneas y saturación de mensajes, además de complicar el seguimiento de feedback y resultados.

Solución aplicada:

Se introduce una variable booleana goal_active que impide enviar un nuevo goal si existe uno en ejecución. Esta variable se activa al enviar el goal y se desactiva cuando se recibe el resultado final.

Fragmentos de código (action_client_node.py):

````
Condición de activación:
```python
if not self.goal_active and distance > 0.55:
    self.send_goal()
```

Cambio de estado al enviar:
```python
self.goal_active = True
```

Liberación de estado al finalizar:
```python
self.goal_active = False
```
````

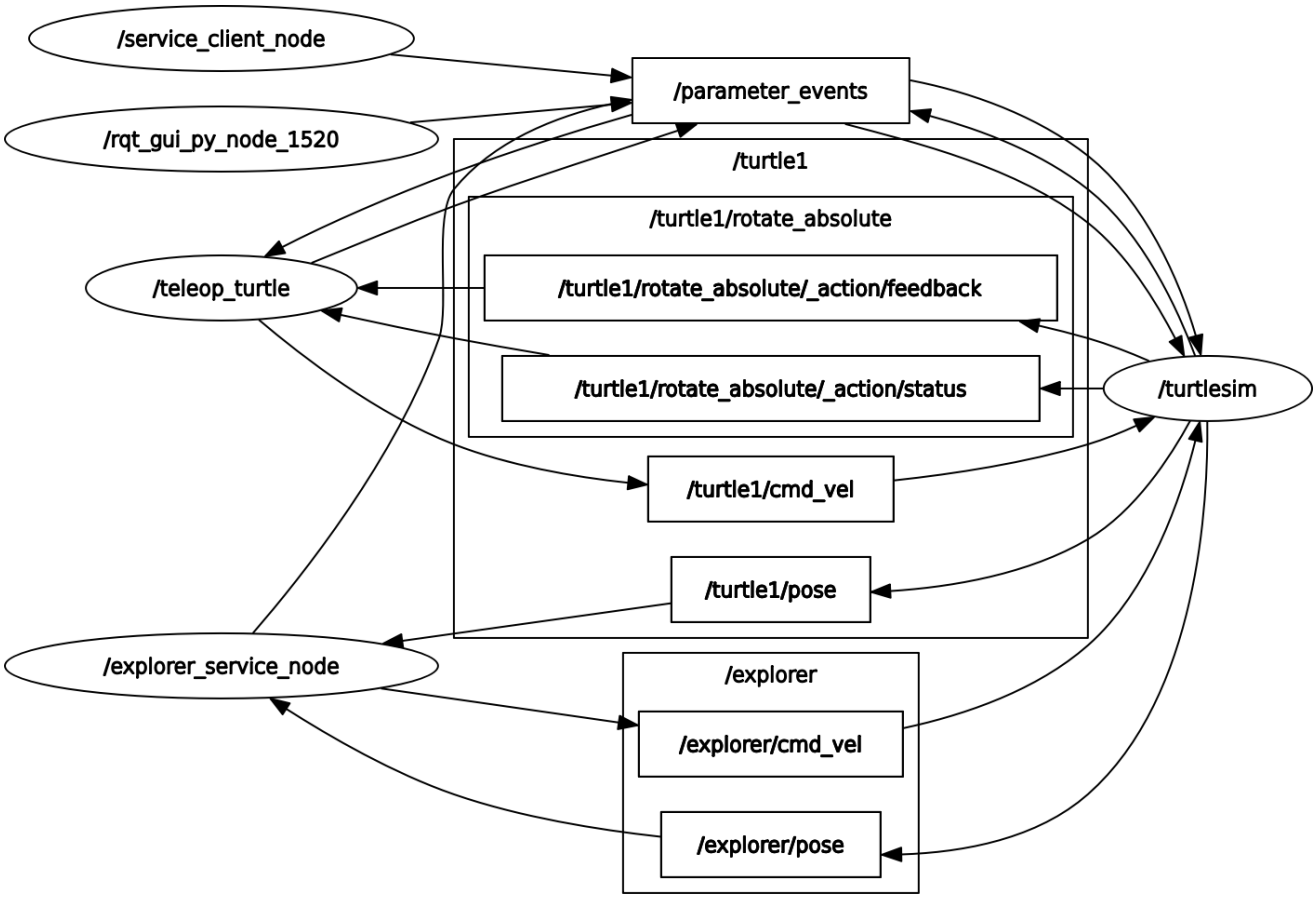
Justificación de la solución:

Este patrón garantiza que el cliente gestiona un único goal activo, lo cual hace el comportamiento más estable, predecible y fácil de evaluar durante pruebas.

8. Mapa de nodos (rqt_graph)

En rqt_graph los óvalos representan nodos y los rectángulos representan comunicaciones (topics y la estructura interna de acciones).

Explicación del mapa de nodos – Service



En la versión Service se observan como nodos principales /turtlesim, /explorer_service_node y /service_client_node (además de nodos auxiliares como /teleop_turtle si se utiliza teclado, /rqt_gui_py_node_... y el topic /parameter_events). El nodo /turtlesim actúa como simulador: publica el estado de las tortugas mediante /turtle1/pose y /explorer/pose, y recibe comandos de velocidad para ejecutar el movimiento.

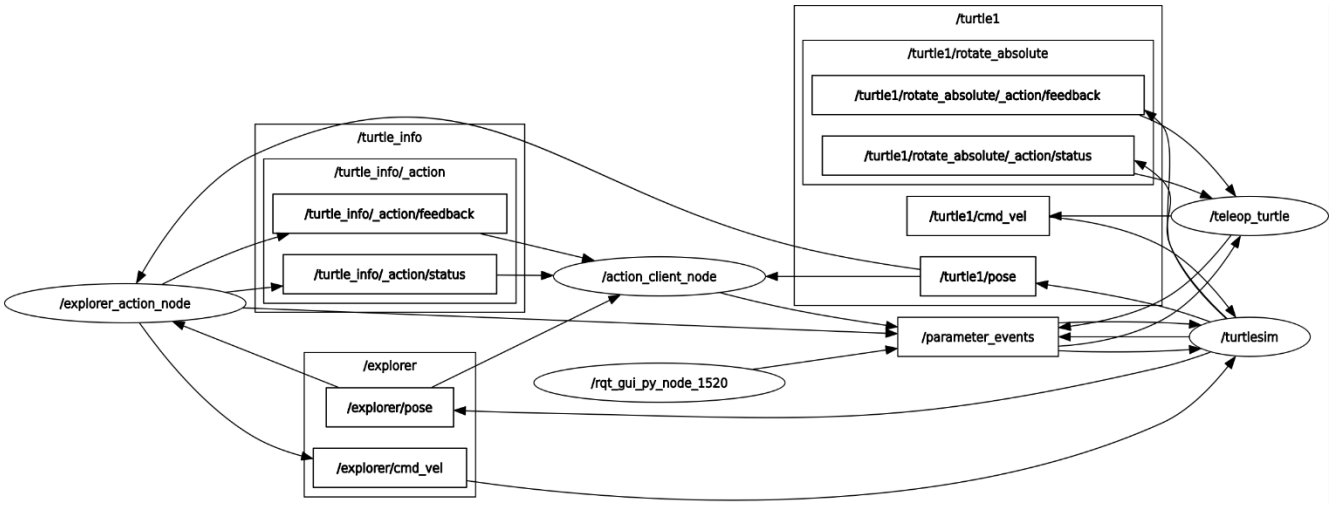
El seguimiento de explorer se aprecia en el ciclo de comunicación por topics. En concreto, /explorer_service_node se suscribe a /turtle1/pose y /explorer/pose para conocer la posición del objetivo y su propia posición. Con esos datos calcula el control proporcional (distancia y error

angular) y publica el comando de velocidad en /explorer/cmd_vel. Dicho comando es consumido por /turtlesim, que actualiza el movimiento y vuelve a publicar las poses, cerrando así el bucle de control.

La característica que diferencia esta versión es la presencia del cliente /service_client_node, cuya función es invocar periódicamente el servicio turtle_info ofrecido por /explorer_service_node. Conceptualmente, el cliente solicita el estado actual (request vacío) y el servidor responde con la información completa (poses, velocidades y distancia). En el grafo, esta relación cliente-servidor se interpreta como consultas puntuales repetidas, adecuadas cuando se desea “leer el estado” de forma periódica sin necesidad de feedback continuo ni una tarea con condición de finalización.

Por último, si se ejecuta /teleop_turtle, se observa el topic /turtle1/cmd_vel, publicado por dicho nodo y consumido por /turtlesim, permitiendo que turtle1 se mueva manualmente y, como consecuencia, el seguimiento por parte de explorer se adapte a esos cambios.

Explicación del mapa de nodos – Action



En la versión Action se observan como nodos principales /turtlesim, /explorer_action_node y /action_client_node (además de los mismos elementos auxiliares: /teleop_turtle si se usa, /rqt_gui_py_node_... y /parameter_events). Al igual que en Service, /turtlesim publica /turtle1/pose y /explorer/pose y consume /explorer/cmd_vel para mover a explorer. De este modo, el bucle de control de seguimiento se mantiene: /explorer_action_node recibe poses, calcula el control proporcional y publica comandos de velocidad para aproximar explorer a turtle1.

La diferencia clave respecto a Service es que, en lugar de un servicio de consulta, aparece la estructura propia de una acción asociada a turtle_info. En ROS 2, una acción se implementa internamente mediante varios canales (por ejemplo, status y feedback), por lo que en el grafo se visualizan elementos como turtle_info/_action/feedback y turtle_info/_action/status. Esta parte del mapa representa la comunicación entre /action_client_node (cliente de acción) y /explorer_action_node (servidor de acción): el cliente envía un goal, el servidor publica feedback periódico con la evolución del estado (por ejemplo, distancia y poses) y finalmente devuelve un resultado cuando se cumple la condición de finalización (en este proyecto, cuando explorer alcanza a turtle1 por debajo de un umbral de distancia).

En resumen, el grafo del modo Action muestra dos flujos simultáneos: (1) el flujo de control por topics para mover a explorer y (2) el flujo de la

acción turtle_info para ofrecer información con feedback continuo y resultado final. Esto justifica el uso de acciones cuando se necesita observar progreso durante la ejecución de una tarea y disponer de un final bien definido, en lugar de consultas puntuales como en la versión Service.

6. Conclusion

Se ha implementado correctamente un sistema de seguimiento en ROS 2 Humble sobre turtlesim, combinando comunicación continua por topics con mecanismos de consulta/ejecución mediante Service y Action. El control proporcional empleado es simple pero efectivo: permite que explorer oriente su rumbo hacia turtle1 y reduzca la distancia de forma progresiva, siendo además fácilmente ajustable mediante las ganancias del controlador.

A nivel de arquitectura ROS 2, el trabajo permite comparar claramente Service y Action. El servicio es adecuado para obtener lecturas instantáneas y periódicas del estado actual, mientras que la acción resulta más apropiada cuando se necesita feedback continuo y un criterio de finalización asociado a una tarea. El uso de parámetros en launch mejora la reproducibilidad de las pruebas, y la gestión de concurrencia mediante callback groups y MultiThreadedExecutor evita bloqueos entre el control continuo y la atención de peticiones, aportando robustez al sistema.