Sistema de Seguimiento de Tortugas con ROS2

Trabajo 2 - EOII - Versión Extendida

Francisco Nortes Novikov
Vicente Burdeus Sánchez

16 de enero de 2026

Índice

1. Introducción

Este documento presenta el proceso completo de desarrollo e implementación de un sistema de seguimiento de tortugas utilizando ROS2. El objetivo principal es crear un sistema donde una tortuga exploradora siga automáticamente a otra tortuga controlada manualmente, implementando servicios y actions para monitorizar el proceso.

Para la implementación exacta del código, se recomienda revisar los archivos fuente proporcionados en el repositorio del proyecto.

2. Proceso de Desarrollo e Implementación

2.1. Paso 1: Exploración de Topics Iniciales

El primer paso fue comprender la estructura de topics del simulador turtlesim. Iniciamos el simulador y exploramos los topics disponibles:

```
ros2 run turtlesim turtlesim_node
ros2 topic list
```

A continuación, identificamos los tipos de datos de los topics relevantes:

```
ros2 topic info /turtle1/pose
ros2 interface show turtlesim/msg/Pose
ros2 topic info /turtle1/cmd_vel
ros2 interface show geometry_msgs/msg/Twist
```

Con esta exploración descubrimos que el nodo de turtlesim publicaba en `/turtle1/pose` y esperaba comandos de velocidad en `/turtle1/cmd_vel`. Para verificar esta hipótesis, realizamos las siguientes pruebas:

```
ros2 topic echo /turtle1/pose
ros2 topic echo /turtle1/cmd_vel
ros2 run turtlesim turtle_teleop_key
ros2 topic echo /turtle1/cmd_vel
```

Mediante estas pruebas confirmamos que el topic `/turtle1/cmd_vel` solo utilizaba las componentes `linear.x` y `angular.z` del mensaje Twist, información crucial para el diseño del controlador.

2.2. Paso 2: Creación del Paquete y Nodo Básico

Creamos el paquete principal **follower** de tipo Python con las dependencias necesarias:

```
ros2 pkg create follower --build-type ament_python \
  --dependencies rclpy geometry_msgs turtlesim
```

Configuramos correctamente los archivos `setup.py` y `package.xml`, y creamos el primer nodo básico capaz de:

- Suscribirse a `/turtle1/pose`
- Publicar comandos de velocidad en `/turtle1/cmd_vel`
- Implementar un controlador proporcional básico

Para el diseño del controlador, utilizamos las siguientes ecuaciones matemáticas:

- **Distancia**: $d = \sqrt{(x_t - x_e)^2 + (y_t - y_e)^2}$
- **Ángulo objetivo**: $\alpha = \text{atan2}(y_t - y_e, x_t - x_e)$
- **Error angular**: $\theta_e = \text{atan2}(\sin(\alpha - \theta_{actual}), \cos(\alpha - \theta_{actual}))$
- **Velocidad lineal**: $v = 1{,}0 \times d \times \cos(\theta_e)$
- **Velocidad angular**: $\omega = 4{,}0 \times \theta_e$

Estas ecuaciones permiten que la tortuga exploradora ajuste tanto su velocidad lineal como angular en función de la distancia y orientación relativa a la tortuga objetivo.

2.3. Paso 3: Creación del Servicio de Información

Para proporcionar información sobre el estado del sistema, creamos un paquete adicional para las interfaces personalizadas:

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

Definimos el servicio `TurtleInfo.srv` con la siguiente estructura:
Request: Vacío (no requiere parámetros de entrada)
Response: Incluye los siguientes campos de tipo `float64`:

- `turtle_x, turtle_y`: Posición de la tortuga objetivo
- `explorer_x, explorer_y`: Posición de la tortuga exploradora
- `turtle_theta, explorer_theta`: Orientación de ambas tortugas
- `turtle_linear_velocity, turtle_angular_velocity`: Velocidades de la tortuga objetivo
- `explorer_linear_velocity, explorer_angular_velocity`: Velocidades de la tortuga exploradora
- `distance`: Distancia entre ambas tortugas

La nomenclatura elegida ("turtle" y "explorer") sigue las convenciones de ROS2 para facilitar la comprensión del código.

Implementamos tanto el servidor del servicio en el nodo principal como un cliente en un nodo separado para las pruebas.

2.4. Paso 4: Preparación para Primeros Tests

Durante esta fase detectamos que faltaba implementar la creación automática de la tortuga exploradora. Añadimos la funcionalidad necesaria que:

- Llama al servicio `/spawn` de turtlesim
- Crea la tortuga "explorer" en una posición aleatoria
- Configura los parámetros necesarios del nodo

Creamos el primer archivo launch (`launch.xml`) siguiendo la documentación oficial de ROS2, que lanzaba:

- El simulador turtlesim
- El nodo principal de seguimiento
- El nodo cliente del servicio de información

Configuramos `setup.py` para incluir el directorio de launch files y procedimos a compilar y probar:

```
colcon build --packages-select follower follower_interfaces
source install/setup.bash
ros2 launch follower launch.xml
```

2.5. Paso 5: Implementación del Action Server

Creamos la interfaz para el action server añadiendo `CatchTurtle.action` al paquete **follower_interfaces**:

Goal: Vacío

Result: `bool caught` (indica si la tortuga fue capturada exitosamente)

Feedback: Los mismos campos que la respuesta del servicio TurtleInfo

La implementación inicial del action server en el nodo principal reveló un problema crítico: el action server bloqueaba la ejecución del nodo, impidiendo que el controlador funcionara correctamente. Este fue el problema más significativo encontrado durante el desarrollo.

2.6. Paso 6: Problema de Bloqueo y Solución

Problema encontrado: Al ejecutar el action server en el mismo nodo que el controlador, el servidor "secuestraba.[el] hilo de ejecución principal, bloqueando todos los demás callbacks.

Soluciones consideradas:

1. Usar `threading.Thread` para ejecutar el action server en un hilo separado
2. Estudiar executors y callback groups de ROS2 para una solución más robusta

Tras investigar la documentación avanzada de ROS2 sobre executors y callback groups, decidimos implementar la segunda opción, que requería una reescritura completa hacia una arquitectura modular.

2.7. Paso 7: Reescritura en Arquitectura Modular

La arquitectura modular final separa completamente las responsabilidades en diferentes nodos:

Módulos core:

- **shared_poses.py**: Clase `SharedPoses` que utiliza `threading.Lock` y `deepcopy` para proporcionar acceso thread-safe a las poses compartidas entre nodos
- **create_explorer.py**: Función auxiliar para crear automáticamente la tortuga exploradora mediante el servicio `/spawn`

Nodos funcionales:

- **pose_savers.py**: Nodo dedicado exclusivamente a suscribirse a `/turtle1/pose` y `/explorer/pose`, almacenando las poses en el objeto compartido
- **explorer_velocity.py**: Nodo con el controlador proporcional que calcula y publica las velocidades en `/explorer/cmd_vel`
- **turtle_info_service.py**: Nodo que implementa el servidor del servicio de información

- **catch_info_action.py**: Nodo que implementa el action server para el proceso de captura
- **follower.py**: Nodo principal que orquesta e inicializa todos los demás nodos

Nodos cliente (para pruebas):

- **follower_server_client.py**: Cliente del servicio de información
- **follower_action_client.py**: Cliente del action server

Esta arquitectura modular resolvió completamente el problema de bloqueo y proporcionó además:

- Mejor separación de responsabilidades
- Código más mantenible y testeable
- Ejecución concurrente sin bloqueos
- Sincronización thread-safe de datos compartidos

2.8. Paso 8: Organización de Launch Files

Para facilitar las pruebas durante el desarrollo, creamos tres archivos launch diferentes:

- **launch.xml**: Sistema básico con solo el seguimiento automático
- **launch_with_server_client.xml**: Añade el cliente del servicio de información
- **launch_with_action_client.xml**: Añade el cliente del action server

Esta organización permitió probar independientemente cada componente del sistema.

2.9. Paso 9: Pruebas Finales y Ajustes

Realizamos pruebas exhaustivas del sistema completo:

1. Verificación del seguimiento básico con diferentes velocidades y trayectorias
2. Pruebas del servicio de información con llamadas concurrentes
3. Validación del action server con múltiples ciclos de captura
4. Pruebas de robustez ante condiciones extremas (movimientos rápidos, cambios bruscos)

Se corrigieron errores menores relacionados con:

- Condiciones de borde en el cálculo de velocidades
- Sincronización de datos en el objeto compartido
- Manejo de errores en las llamadas a servicios

3. Descripción Técnica de la Implementación

3.1. Arquitectura del Sistema

El sistema final consta de dos paquetes ROS2:
Paquete follower:

- Tipo: ament_python

- Dependencias: rclpy, geometry_msgs, turtlesim, follower_interfaces
- Contiene todos los nodos funcionales y clientes

Paquete follower_interfaces:

- Tipo: ament_cmake
- Contiene las definiciones de servicio (TurtleInfo.srv) y action (CatchTurtle.action)

3.2. Controlador Proporcional

El controlador implementado es de tipo proporcional (P) que:

1. Calcula la distancia euclidiana entre las tortugas
2. Determina el ángulo objetivo hacia la tortuga a seguir
3. Calcula el error angular respecto a la orientación actual
4. Ajusta la velocidad lineal proporcionalmente a la distancia, reducida por el coseno del error angular
5. Ajusta la velocidad angular proporcionalmente al error angular

Las constantes proporcionales (1.0 para lineal, 4.0 para angular) fueron ajustadas empíricamente para obtener un seguimiento suave y preciso.

3.3. Sincronización Thread-Safe

La clase `SharedPoses` implementa el patrón de diseño Thread-Safe Singleton mediante:

- `threading.Lock` para proteger el acceso concurrente
- `copy.deepcopy` para devolver copias independientes de los datos
- Métodos getter y setter protegidos por el lock

Esto garantiza que múltiples nodos puedan acceder a las poses sin condiciones de carrera.

4. Interfaces Implementados

4.1. Servicio TurtleInfo.srv

Propósito: Proporcionar información completa sobre el estado actual de ambas tortugas.

Request: Vacío (no requiere parámetros)

Response:

```
float64 turtle_x
float64 turtle_y
float64 explorer_x
float64 explorer_y
float64 turtle_theta
float64 explorer_theta
float64 turtle_linear_velocity
float64 turtle_angular_velocity
float64 explorer_linear_velocity
float64 explorer_angular_velocity
float64 distance
```

4.2. Action CatchTurtle.action

Propósito: Monitorizar el proceso de captura de la tortuga objetivo, proporcionando feedback continuo hasta la captura.

Goal: Vacío (inicia el proceso de captura)

Result:

```
bool caught
```

Feedback: Los mismos campos que TurtleInfo.srv, permitiendo monitorización en tiempo real

5. Problemas Encontrados y Soluciones

5.1. Bloqueo del Action Server

Problema: El action server bloqueaba el nodo principal al ejecutarse en el mismo hilo que los demás callbacks.

Solución: Reescritura completa en arquitectura modular con nodos separados, cada uno con su propio executor. Uso de `SharedPoses` para compartir datos de forma thread-safe entre nodos.

5.2. Sincronización de Datos

Problema: Múltiples nodos necesitaban acceso concurrente a las poses de las tortugas.

Solución: Implementación de `SharedPoses` con `threading.Lock` y `deepcopy` para garantizar acceso thread-safe.

5.3. Creación Automática de la Tortuga Exploradora

Problema: Inicialmente la tortuga exploradora debía crearse manualmente.

Solución: Implementación de función auxiliar que llama al servicio `/spawn` automáticamente al iniciar el sistema.

6. Resultados de Pruebas

6.1. Pruebas Funcionales

Se realizaron las siguientes pruebas con resultados exitosos:

1. **Seguimiento básico:** La tortuga exploradora sigue correctamente a turtle1 en todo momento
2. **Servicio de información:** El cliente recibe datos precisos y actualizados
3. **Action server:** Proporciona feedback continuo y detecta correctamente la captura
4. **Robustez:** El sistema funciona correctamente con movimientos rápidos y cambios bruscos de dirección

Las siguientes figuras muestran el sistema en funcionamiento:

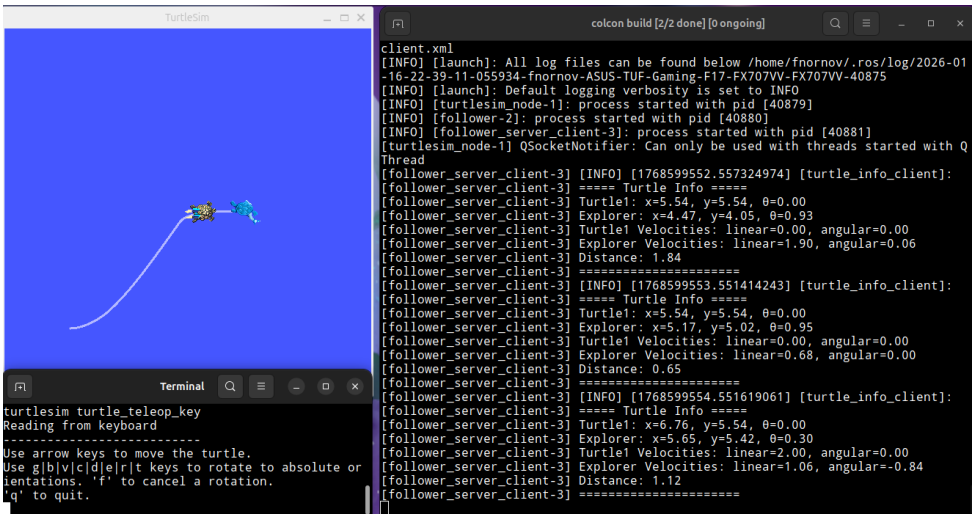

Figura 1: Cliente del servicio de información mostrando datos en tiempo real

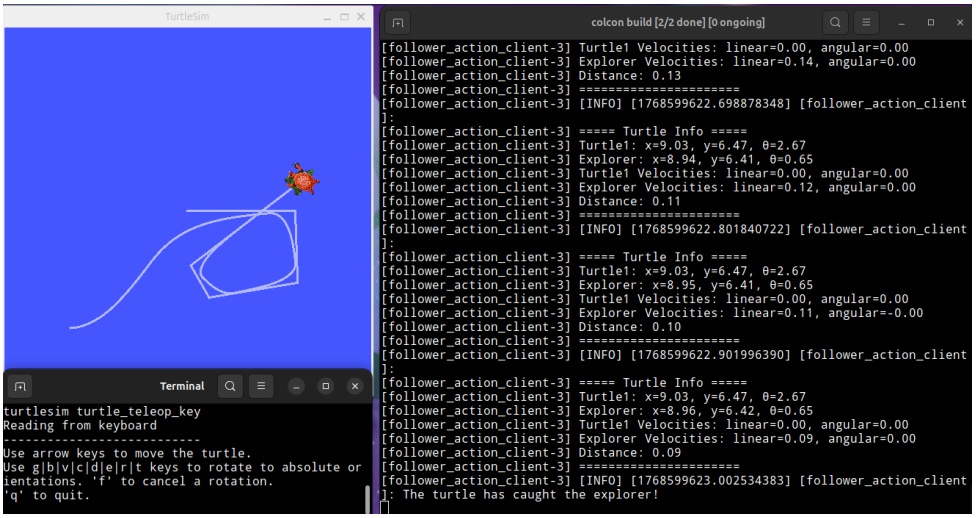

Figura 2: Cliente del action server mostrando feedback del proceso de captura

7. Mapa de Nodos, Topics y Servicios

El siguiente grafo generado con `rqt_graph` muestra la arquitectura completa del sistema, incluyendo todos los nodos, topics, servicios y actions:

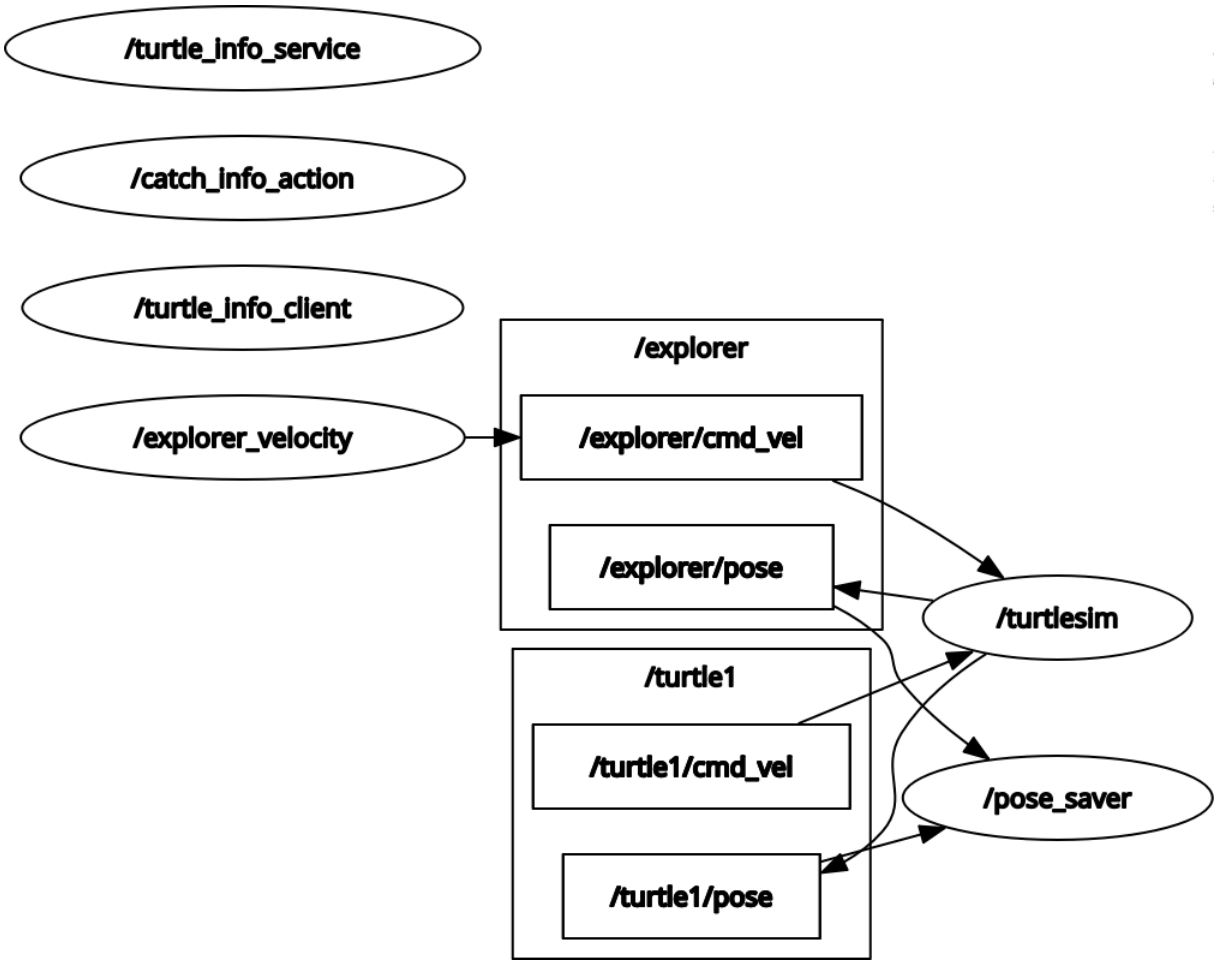

Figura 3: Grafo completo del sistema generado con rqt_graph

El grafo muestra claramente:

- Nodos de turtlesim y teleop
- Nodos del sistema follower
- Topics de poses y comandos de velocidad
- Servicios de información
- Action server de captura

8. Conclusiones

El sistema de seguimiento de tortugas fue implementado exitosamente, cumpliendo todos los requisitos especificados. Los aspectos más destacables incluyen:

- **Arquitectura modular:** Facilita el mantenimiento y la extensibilidad
- **Sincronización robusta:** Garantiza la integridad de los datos compartidos
- **Interfaces bien diseñados:** Permiten monitorización completa del sistema
- **Controlador eficiente:** Proporciona seguimiento suave y preciso

El principal desafío técnico fue la gestión de la concurrencia y el bloqueo del action server, que se resolvió mediante una reescritura completa hacia una arquitectura más

modular y profesional. Esta experiencia demuestra la importancia de comprender los conceptos avanzados de ROS2 como executors y callback groups.