Navegación y Control Interactivo del TurtleBot en ROS2: Un Enfoque Práctico

Brayan Salcedo
Departamento de Ingeniería Eléctrica y Electrónica
Universidad de los Andes
Bogotá D.C, Colombia
b.salcedo@uniandes.edu.co

Edward Manosalva
Departamento de Ingeniería Eléctrica y Electrónica
Universidad de los Andes
Bogotá D.C, Colombia
e.manosalva@uniandes.edu.co

Daniel Contreras
Departamento de Ingeniería Eléctrica y Electrónica
Universidad de los Andes
Bogotá D.C, Colombia
de.contreras@uniandes.edu.co

Santiago Solano
Departamento de Ingeniería Eléctrica y Electrónica
Universidad de los Andes
Bogotá D.C, Colombia
s.solanor@uniandes.edu.co

***Abstract*—Este documento presenta la implementación de una serie de nodos utilizando el ambiente de ROS2 y el programa de simulación de Coppelia para controlar un TurtleBot. Tres problemas grandes son analizados: cómo tele operar el robot por interacción de usuario, cómo mostrar una interfaz gráfica con información relevante del robot en tiempo real, y cómo reproducir una ruta predefinida en un robot. Para ello se recurren a las herramientas de grafos RQT de ROS2 y diagramas de flujo que explican la interacción entre rutinas.**

***Keywords*—ROS2, Coppelia, nodo, tópico, publisher, subscriber, servicio, grafo RQT, TurtleBot.**

I. Introducción

Para este trabajo se utilizaron las versiones de ROS2 Irwini y el software CoppeliaSim utilizando un robot diferencial TurtleBot2 montado sobre una base Kuboki.

La primera parte del proyecto consistió en crear un nodo de ROS2 con el nombre */turtle_bot_teleop*, el cual permite que un usuario manipule el robot a través de la interacción con el teclado. Además, se incluye la capacidad de seleccionar las velocidades angulares y lineales.

Posteriormente, un segundo nodo llamado *turtle_bot_interface* es programado para permitir la visualización de la posición en tiempo real del robot en Coppelia por medio de una interfaz gráfica. Dicha interfaz es capaz de guardar recorrido desde el momento que se inicia la simulación, y le permite al usuario guardar la gráfica con un nombre y en un directorio deseado. Asimismo, el usuario puede decidir si desea guardar el recorrido del robot. En caso afirmativo, la secuencia de velocidades del robot se registra en un archivo de texto (.txt).

Finalmente, se crea un nodo llamado */turtle_bot_player* que con el archivo de texto mencionado anteriormente, es capaz de reproducir la secuencia del robot. Esto se genera por medio de un servicio que conecta con la interfaz creada anteriormente.

II. Teleoperación del robot

Se crea un nodo con el nombre de */turtle_bot_teleop* siguiendo la lógica presentada en el diagrama de flujo de la Figura 1. La función de este nodo consiste en permitir al usuario manejar el movimiento del robot a través de la interacción con el teclado. Para ello, inicialmente el nodo le pregunta al usuario una velocidad lineal y una velocidad angular en *cm/s* y *deg/s* para mover al robot. Es decir, el usuario tiene la libertad de ajustar las velocidades con que controla el robot, tal que dichas cantidades no se encuentran preestablecidas dentro del código y el usuario tampoco necesita cambiar el código para modificar la velocidad del robot. En la Figura 2 se muestran las ventanas emergentes que aparecen al inicio para que el usuario digite esta información.

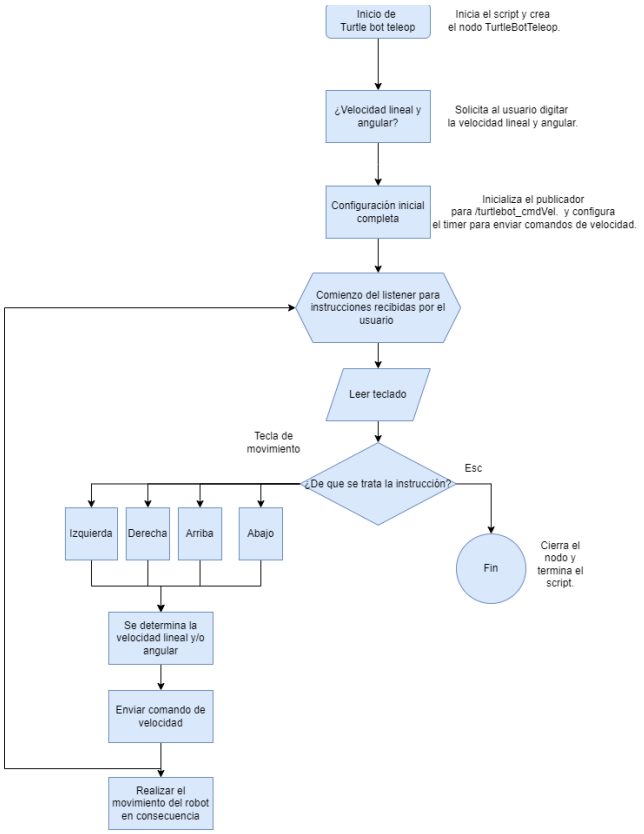


Fig. 1. Diagrama de flujo para el funcionamiento del nodo */turtle_bot_teleop*.

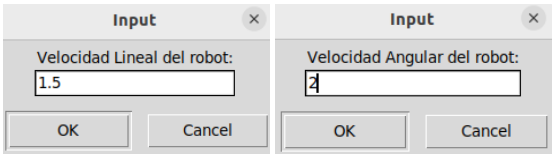


Fig. 2. Ventanas emergentes para preguntar por velocidad lineal y angular.

La pequeña interfaz de la Figura 2 se genera utilizando la librería de Python *tkinter*. En este caso se trata de dos ventanas que aparecen de forma secuencial configuradas para recibir como parámetro un número de punto flotante. Una vez el usuario define dichas cantidades, están son guardadas dentro de la clase definida dentro del nodo */turtle_bot_teleop* tal que todos los movimientos posteriores se realizan usando dichas velocidades. La asignación de dichas cantidades se denominará como *self.linear_speed* y *self.angular_speed* respectivamente. Adicionalmente, en caso de que el usuario de forma incorrecta cierre las ventanas o presione cancelar, en el nodo se programaron unas velocidades por defecto para evitar errores en la ejecución; y dichos valores fueron configurados en $1.5\ cm/s$ y $2.0\ deg/s$.

Luego, para realizar el proceso de escucha del teclado (*listener*), se utilizó la librería *pynput* y se utilizó un método de captura de eventos del teclado. Se definieron entonces eventos para 4 teclas: las flechas de arriba, abajo, izquierda y derecha. Según la tecla escogida los comandos enviados posteriormente al robot se sintetizan en la Tabla I (definición de posibles acciones). Adicionalmente, cuando se deja de presionar alguna de estas teclas, inmediatamente se envía una velocidad de 0 al robot para que frene su movimiento. Lo mismo ocurre cada 0.15 segundos en el caso que no se detecte ninguna tecla presionada después de haber iniciado el movimiento. Finalmente, en cualquier momento el usuario puede presionar la tecla ESC para detener el proceso de captura de eventos y finalizar así la interacción con el robot.

TABLA I. CONJUNTO DE VELOCIDADES ENVIADAS SEGÚN LA TECLA PRESIONADA.

Tecla presionada	Velocidad lineal [*cm/s*]	Velocidad angular [*deg/s*]
Flecha arriba (*Key.up*)	+self.linear_speed	0
Flecha abajo (*Key.down*)	-self.linear_speed	0
Flecha izquierda (*Key.left*)	0	+self.angular_speed
Flecha derecha (*Key.right*)	0	-self.angular_speed
Soltar una de las teclas anteriores	0	0
No presionar una tecla en 0.15 segundos	0	0
Escape (ESC)	Detiene la escucha del teclado	

Finalmente, hace falta definir cómo el nodo realiza el envío de estos comandos al TurtleBot dentro de Coppelia. En el entorno de ROS2, esta tarea se cumple mediante la creación de un *publisher* al tópico /*turtlebot_cmdVel*. Esto significa que el nodo */turtle_bot_teleop* publica información a un tópico que es libre para leer por otros nodos, incluyendo el entorno de simulación. Dichas publicaciones se hacen utilizando una profundidad de QOS (*QOS history window*) de 50, mayor a la estándar de 10, porque debido a la latencia del teclado y el tiempo de muestreo del hardware y software del computador es preferible dedicar una mayor ventana a reintentar enviar mensajes que se juntan demasiado rápido. Para visualizar el problema de nodos y tópicos, se hace uso de un *rqt_graph* de ROS2, obtenido durante la ejecución de */turtle_bot_teleop* dentro de la aplicación diseñada.

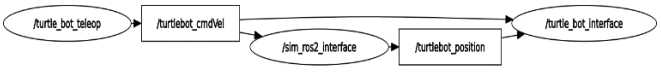


Fig. 3. Grafo RQT generado por ROS2 sobre los nodos y tópicos cuando se ejecuta */turtle_bot_teleop* en la aplicación diseñada.

Se evidencia claramente en la parte izquierda de la Figura 3 cómo el nodo (encerrados en óvalos) *\turtle_bot_teleop* publica información en un tópico (encerrados en rectángulos) /*turtlebot_cmdVel*. En este tópico se publican mensajes de velocidad con formato *Twist*, que corresponde a dos vectores en tres dimensiones (*x, y, z*) de velocidad lineal y angular. Luego cualquier nodo que se *suscriba* a dicho tópico puede leer la información publicada. En este caso se evidencia cómo hay dos nodos suscritos a */turtlebot_cmdVel: /turtle_bot_interface* y */sim_ros2_interface*, pero se hablará más sobre ellos en la siguiente sección del documento.

Finalmente, para mostrar el correcto funcionamiento del nodo *\turtle_bot_teleop*, se adjunta en la Figura 4 una captura de pantalla donde se evidencia cómo en Coppelia se están recibiendo los respectivos comandos de velocidad con las cantidades definidas en la Figura 2. En la misma captura se muestra una terminal donde tras ejecutar el comando el comando ros2 topic echo */turtlebot_cmdVel* se imprimen los mensajes publicados al tópico con formato de mensaje Twist.

Fig. 4. Monitoreo de los mensajes publicados en el tópico */turtlebot_cmdVel* por el nodo */turtle_bot_teleop*.

III. INTERFAZ DE USUARIO

Se crea una interfaz utilizando el nodo */turtle_bot_interface* con tres propósitos:

- Englobar el acceso a los nodos */turtle_bot_teleop* y */turtle_bot_player*
- Mostrar una gráfica en tiempo real de la posición del robot en el entorno de simulación y su trayectoria de movimiento.
- Guardar un recorrido del usuario como un archivo de texto para ser replicado posteriormente.

El proceso lógico de la interfaz se muestra en el diagrama de flujo de la Figura 5:

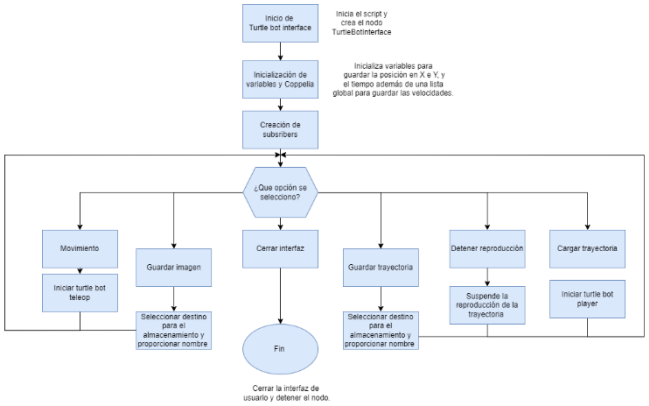


Fig. 5. Diagrama de flujo para el funcionamiento del nodo */turtle_bot_interface*.

La creación de la interfaz gráfica que cumple con estos objetivos se hace utilizando la librería de Python *PyQt5*. Esta fue seleccionada por su capacidad de incluir gráficas, botones, mensajes de texto, celdas de entrada y ventanas emergentes al mismo tiempo, siguiendo además una actualización en tiempo real de información. Aunque no se profundiza en todas las librerías utilizadas para estas distintas tareas, vale la pena resaltar que la capacidad de mantener la interfaz abierta y en constante actualización al mismo tiempo

que de fondo se ejecuta un nodo como */turtle_bot_teleop* o */turtle_bot_player* se debe gracias al uso de *threading*. Esto quiere decir que se crean distintos hilos de forma paralela para ejecutar los otros nodos, que contrasta con la lógica secuencial que tienen otras librerías como *tkinter*.

Al comienzo de la ejecución del nodo, se presenta la interfaz mostrada en la Figura 6. Esta primera pantalla contiene dos botones: el primero resulta en la ejecución de */turtle_bot_teleop* (a lo cual sigue la lógica explicada en la sección II) y la segunda resulta en la ejecución de */turtle_bot_player* (resultando en la lógica de la sección IV).

Fig. 6. Primera pantalla de la interfaz obtenida al ejecutar */turtle_bot_interface*.

Después de la aparición de la ventana de la Figura 6 y de haber cargado la escena de simulación en Coppelia, el grafo RQT que se obtiene a través de ROS2 se muestra en la Figura 7. En este caso se evidencia una creación de dos *suscripciones* en el nodo: una al tópico */turtlebot_cmdVel* y otra al tópico */turtlebot_position*.

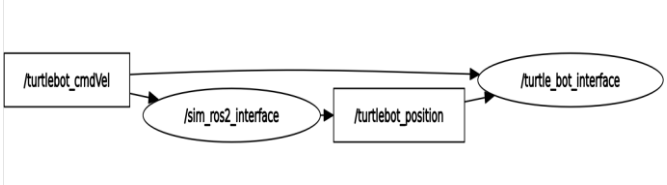


Fig. 7. Grafo RQT generado por ROS2 sobre los nodos y tópicos cuando se ejecuta */turtle_bot_interface* y se carga la escena en Coppelia.

Se observa en la Figura 7 entonces que */turtle_bot_interface* recibe información que se encuentre en los tópicos de velocidad y posición. La razón para suscribirse al tópico con los mensajes de velocidad es para eventualmente poder guardar los comandos de velocidad publicados en un archivo de texto que se pueda replicar en el futuro. Y la suscripción al nodo de posición es para graficar la posición actual del robot y trazar su trayectoria dentro de la interfaz. Nótese que la única diferencia con la Figura 3 y Figura 14 es que en estos grafos existe un *publisher* al tópico de */turtlebot_cmdVel*, respectivamente */turtle_bot_teleop* o */turtle_bot_player*, pero mientras la interfaz continua en su estado de la Figura 6, dichos nodos no habrían sido creados todavía.

Inicialmente se explicará la interfaz obtenida cuando se escoge la tele operación. Al presionar el botón aparece una ventana como la de la Figura 8, al mismo tiempo que se ejecuta toda la rutina detallada previamente para */turtle_bot_teleop* (para este punto el grafo RQT es el mismo mostrado en la Figura 3). Esta interfaz cuenta con 4 partes principales: un espacio para graficar la posición y trayectoria del robot, un espacio para introducir un nombre a la gráfica, un botón para guardar la gráfica como un archivo de imagen, y un botón para guardar la serie de comando de velocidad dentro de un archivo de texto.

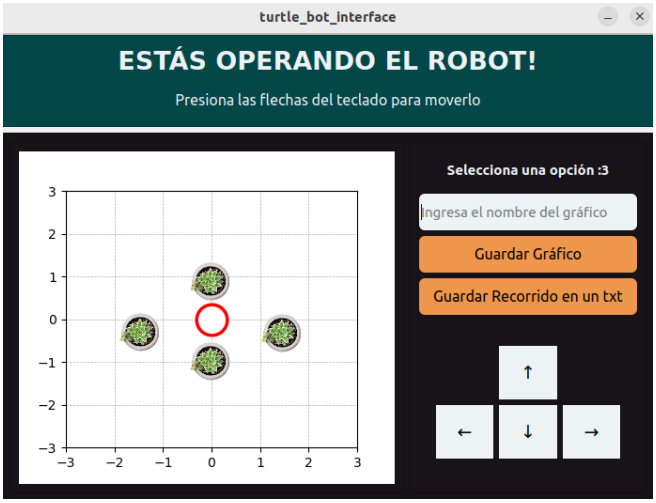


Fig. 8. Ventana de la interfaz obtenida al seleccionar la teleoperación.

La Figura 9 muestra un ejemplo de cómo cambia la gráfica a medida que se muestra el robot, cómo se puede cambiar el nombre de la gráfica, y la ventana adicional que emerge para guardar la imagen o archivo de texto.

Fig. 9. Evolución normal de la interfaz a medida que mueve al robot por tele operación.

La gráfica mostrada se genera utilizando las librerías de *matplotlib*. En el diagrama la línea negra corresponde a la trayectoria que ha seguido el robot desde su inicio en el centro del tablero, mientras que el círculo rojo encierra su posición actual. Adicional a eso, se ubicaron en la gráfica imágenes de plantas que corresponden a los obstáculos de la simulación. Como se mencionó previamente, esta gráfica se construye a partir de los datos observado en el tópico */turtlebot_position*, a partir del cual se construyen listas con todas las posiciones *x* e *y* a lo largo del tiempo para graficar la trayectoria. Y con el objetivo de no sobrecargar el programa y mostrar fluidez en pantalla, se escogió que la gráfica se actualizaría cada 10 milisegundos. Por último, se comprueba cómo el título de la gráfica puede ser cambiado por el usuario.

Por último, es importante detallar la creación del archivo de texto con el recorrido. Como muestra la Figura 7, al estar suscrito el nodo */turtle_bot_interface* al tópico */turtlebot_cmdVel*, se programa que el nodo al leer cada uno de los comandos extraiga 4 variables: el tiempo en que se

publicó el mensaje (el cual puede ser obtenido utilizando la librería *time()*), el componente *x* de la velocidad lineal, el componente *y* de la velocidad linear, y finalmente la componente z de la velocidad angular. Estas 4 variables por comando son luego indexadas y agregadas a una lista. Cuando el usuario presiona el botón y selecciona una ruta, el nodo escribe un archivo de texto con la lista guardada, transcribiendo línea por línea comandos de velocidad con el siguiente formato:

$$Velocity: [time, linear\_x, linear\_y, angular\_z]$$

En la Figura 10 se muestra una captura de pantalla de un archivo de texto creado para el recorrido de la Figura 9, donde se evidencia el anterior formato.

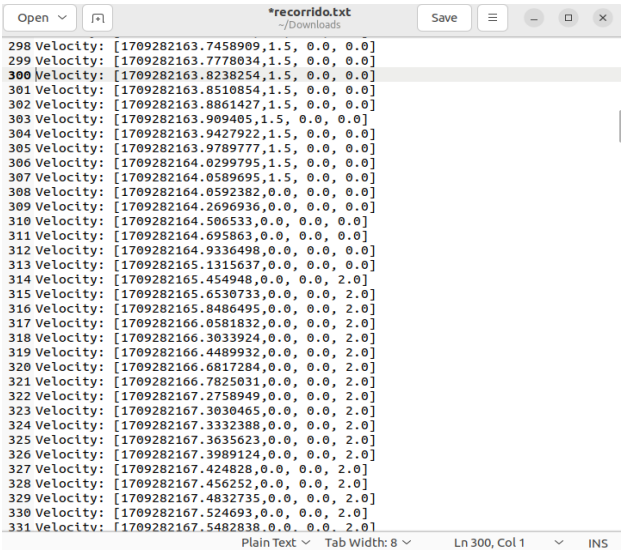


Fig. 10. Formato de un archivo de texto creado con las velocidades leídas en el tópico.

Nótese que en teoría los tiempos de este archivo deberían coincidir con los tiempos en que el robot de Coppelia ejecuta los comandos porque corresponden al tiempo que ve otro nodo suscrito al mismo tópico.

En el otro caso, donde el usuario escoge cargar una ruta predefinida, a partir de la ventana inicial de la Figura 6 se abre una ventana emergente donde el usuario puede navegar las carpetas del computador y escoger el archivo de texto con el recorrido guardado. Esto se muestra en la Figura 11. Nótese en esta Figura donde la ventana emergente dice *Open* en lugar de *Save as* como ocurría en la Figura 9.

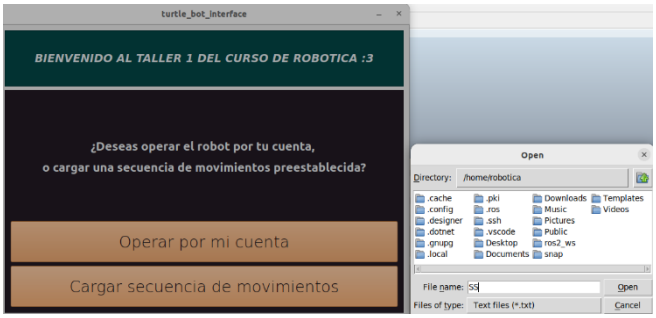


Fig. 11. Cuando el usuario escoge cargar una secuencia se abre una ventana emergente para seleccionar un archivo de texto.

Inmediatamente cuando el usuario carga el archivo, la ventana de la interfaz cambia y el robot comienza a moverse según la secuencia. La interfaz que se abre es muy similar, únicamente cambiando los textos y removiendo el botón de guardar el recorrido como archivo de texto. La Figura 12 muestra cómo aparece una gráfica con espacio para un nombre y exportar la imagen.

Fig. 12. Ventana de la interfaz obtenida al seleccionar cargar secuencia.

La lógica del proceso que se ejecuta para la lectura y envío de los comandos se detalla a profundidad en la siguiente sección.

IV. Reproducción de ruta

Como se mostró previamente en la Figura 11 y 12, el proceso de selección de archivo es algo que ocurre a partir del nodo */turtle_bot_interface*, pero la lectura y envío de los comandos se realiza en el nodo */turtle_bot_player*. Este último nodo fue creado a partir de la lógica que aparece en el siguiente diagrama de flujo:

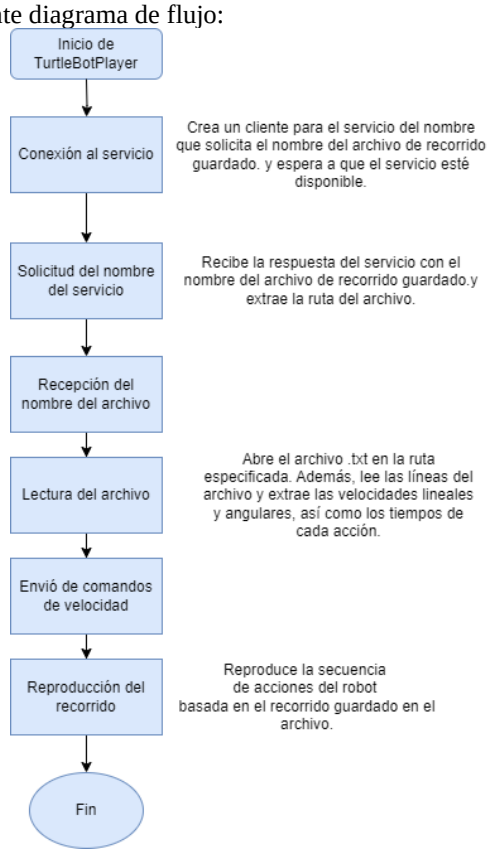


Fig. 13. Diagrama de flujo para el funcionamiento del nodo */turtle_bot_player*.

Como se muestra en la Figura 13, el intercambio de información a través del cual el nodo */turtle_bot_player* recibe el nombre del archivo que debe leer se realiza a través de un *servicio*. Para ello, fue necesario en la implementación crear un servicio personalizado para enviar esta información. En la implementación entonces se creó un segundo paquete aparte de */turtle_bot_2*, llamado */my_robot_interfaces*. Este paquete de C++ contiene en una carpeta *\srv* la declaración del tipo de servicio */Nombre*, que recibe como *request* una variable de tipo *string* (*a*) y devuelve otro *string* (*Nombre*) con la ruta del archivo seleccionado en la interfaz.

En este orden de ideas, justamente cuando el usuario presiona el botón de cargar una secuencia y cambia la interfaz como en la Figura 11, detrás se ha creado el nodo */turtle_bot_player*¸ el cual a su vez crea un *publisher* a */turtlebot_cmdVel y* crea un servicio de tipo */Nombre* bajo la denominación */txt_name*. El nodo (que es un cliente) luego envía al servicio una *request* con el mensaje "SolicitandoNombreDelArchivo". Este mensaje en realidad no es relevante, porque el servicio únicamente se configuró para retornar el nombre del archivo seleccionado independientemente de la *request*, únicamente funciona como un medio de confirmación. El mensaje con la ruta del archivo de texto se recibe justamente después que el usuario selecciona el archivo, y en el tiempo intermedio que hay entre ambos sucesos, cada un segundo el nodo /turtle_bot_player publica en la terminal el mensaje de advertencia: "Waiting for Service".

Por ejemplo, el mensaje *string* que recibiría el nodo */turtle_bot_player* utilizando la instancia *future*, tendría el siguiente formato:

$$my\_robot\_interfaces.srv.Nombre\_Response(name = 'route')$$

Donde *route* es la ruta del archivo dependiente de la selección del usuario. Así, el nodo */turtle_bot_player* extrae esta ruta del mensaje y posteriormente comienza el proceso de lectura línea por línea, y las publica en */turtlebot_cmdVel*. En este caso, el publisher no se configura distinto al */turtle_bot_teleop*, es decir, se siguen mandando mensajes de tipo *Twist* con una ventana QOS de 50. La única diferencia es que en este caso al leer todas las líneas del archivo de texto, se toma lectura de todos los tiempos que habían sido indexados en la primera columna del mensaje como se muestra en la Figura 10, y el nodo utiliza estos tiempos para esperar entre el envío de un mensaje y otro. Concretamente, se toma la diferencia entre dos mensajes consecutivos para enviar dichos mensajes, y dicho tiempo se recalcula para cada mensaje. No obstante, se incluyó como un factor de corrección multiplicar este intervalo de tiempo por un 88%, debido a que se encontró que el tiempo que le tomaba al nodo publicar el mensaje y que el robot lo leyera sería mayor al intervalo especificado por la demora de procesos internos.

Podemos confirmar en el grafo de la Figura 14 la conexión entre tópicos:

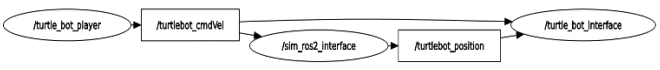
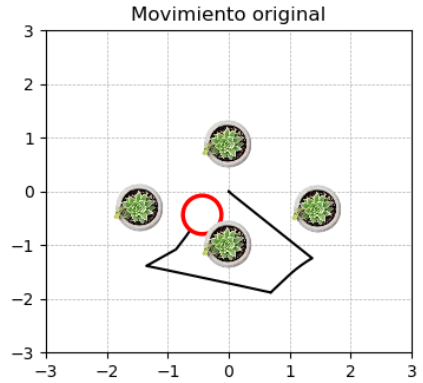


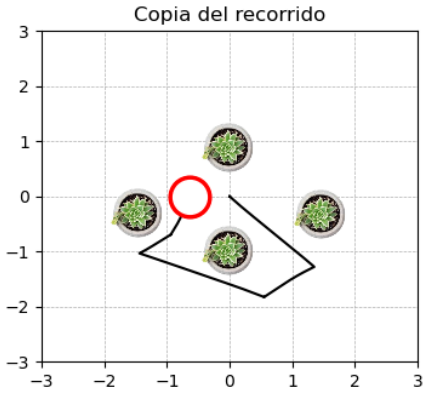
Fig. 14. Grafo RQT generado por ROS2 sobre los nodos y tópicos cuando se ejecuta */turtle_bot_player* en la aplicación diseñada.

Este grafo es igual al grafo de la Figura 3, únicamente reemplazando el nodo */turtle_bot_teleop* por el nodo */turtle_bot_player*. Es decir, se evidencia a través de esta representación como la función de */turtle_bot_player* es la misma que */turtle_bot_teleop* desde la perspectiva del simulador, ya que ambos son nodos que simplemente publican mensajes en el tópico /*turtlebot_cmdVel*.

Con esta implementación, se consigue reproducir de manera fiel y satisfactoria una secuencia de movimientos como se muestra en el ejemplo de la Figura 15, donde la imagen (*a*) corresponde a un movimiento controlado por el usuario, y (*b*) es la reproducción a partir del archivo de texto generado.



(*a*)



(*b*)

Fig. 15. Gráfica obtenida con (*a*) */turtle_bot_teleop* en una interacción el usuario y (*b*) reproducción del mismo recorrido a partir del archivo de texto generado utilizando el nodo */turtle_bot_player*.

Se evidencia cómo la Figura 15*b* es una copia cercana y fiel al movimiento original mostrado en la Figura 15*a*, demostrando así el correcto funcionamiento de */turtle_bot_player*. No obstante, siempre van a haber pequeñas diferencias, porque el simulador no es ideal y por ende la inercia del robot puede afectar los resultados de manera significativa. Existen dos estrategias para que el movimiento de la reproducción sea más fiel a pesar de este problema. La primera estrategia es manejar un rango de velocidades bajas, ya que así se reduce la inercia que puede tener el robot en momentos de máxima velocidad. Concretamente, se ha encontrado que los valores de en $1.5\ cm/s$ y $2.0\ deg/s$ (linear y angular) resultan encontrase en un rango apropiado para este fin. En segundo lugar, el usuario debería esperar a que el robot esté completamente quieto antes de enviar un comando de movimiento distinto, ya que de lo contrario se distorsionaría el movimiento precisamente por la inercia que queda como residuo en otra dirección.

V. Conclusiones

A lo largo del documento se presentó una explicación y evidencia de cómo se puede lograr una implementación exitosa en el entorno de ROS2 de operaciones en interacción con un simulador de robótica. Concretamente se lograron apreciar las ventajas que tiene ROS2 para englobar el ambiente de trabajo y comunicar distintas rutinas programadas en un lenguaje común como puede ser Python.

A través de los grafos de las Figuras 3, 7 y 14, se evidencia cómo el envío y recibo de mensajes se puede lograr de manera efectiva y eficiente a través de tópicos dedicados a un tipo de mensaje. Especialmente se evidenció como en una misma interacción con el simulador se pueden alternar dos nodos distintos, como lo eran */turtle_bot_teleop* y */turtle_bot_player*, y conseguir de manera objetiva los mismos resultados aunque el proceso haya sido distinto. Esto en otro ambiente sin tópicos habría sido más complicado porque se habría tenido que reestructurar la comunicación. Del mismo modo, se logró observar las diferencias cruciales entre *publishers* y *subscribers*. En este caso resultó que fue muy práctico utilizar un único *subscriber* común (*/turtle_bot_interface*) y a partir de la información recibida realizar operaciones relacionadas directamente con la experiencia de usuario y con el entorno del computador.

Adicionalmente, el servicio muestra ser también un método efectivo para recibir y enviar información. Especialmente resulta útil para sobrellevar la asincronía que puede existir entre dos nodos distintos, problema que no es tan fácil de solucionar en el caso de los tópicos. Es decir, el servicio puede estar caído, pero cuando se habilite podrá enviar la información al nodo que la solicita, o, al contrario, el servicio puede estar listo para enviar la información, pero el receptor todavía no hace una *request*, y es cuando los dos interactúan en cierto momento específico que se da la comunicación. Por otro lado, en el tópico surgía el problema que si se comenzaban a publicar mensajes y el *subscriber* no estaba listo para leerlos, mucha información se perdía. Incluso el servicio programado puede ser mejorado diseñando respuestas distintas para *requests* distintas, lo cual es ideal cuando un gran número de clientes se conecta al mismo servicio, que no era el caso en esta práctica.

Por último, se evidenciaron dos grandes retos en la implementación. El primero fue sobre cómo generar una interfaz gráfica que permitiera la ejecución de varios nodos a la vez y al mismo tiempo brindar una buena experiencia al usuario. La solución encontrada a este problema fue el uso de *threading*, con lo cual se puede lograr una ejecución en paralelo de distintas rutinas sin que estas se interrumpan entre sí y sin necesariamente seguir una lógica estrictamente secuencial en la interfaz. Y en segundo lugar, se encontró que reproducir de forma exacta una secuencia de movimientos a partir de “control” de velocidad puede resultar muy difícil o imposible en un robot debido a los problemas físicos de inercia que se pueden presentar. Las únicas soluciones encontradas fueron el uso de velocidades bajas y una operación pausada para no generar movimientos disruptivos. Una programación más exhaustiva y compleja podría hacerse analizando la velocidad de cada una de las llantas para así lograr un control preciso en aplicaciones de más alto nivel.