ROS2 Apuntes

¿Qué es ROS2, cuándo usarlo y por qué?

Ros significa "sistema operativo de robots". Es algo entre un middleware y un marco construido especialmente para aplicaciones de robótica. El objetivo de Ros2 es proporcionar un estándar para el software de robótica que los desarrolladores pueden usar y reutilizar en cualquier robot.

¿Qué tal si pudieras construir sobre una base de software sólida para programar directamente funcionalidades de alto nivel y trabajar en casos de uso en lugar de cuestiones técnicas de bajo nivel que ya existen de todos modos? Este es el por qué detrás de Ros.

Ros2 está aquí para ayudarte a desarrollar aplicaciones robóticas potentes y escalables. Puedes usarlo siempre que desees crear un software para un robot que requiere mucha comunicación entre sus subprogramas o tiene funcionalidades que van más allá de un caso de uso muy simple.

Estos son los dos puntos principales que describen a ROS2:

1. En primer lugar, Ros2 te proporciona una forma de separar tu código en bloques reutilizables, junto con un conjunto de herramientas de comunicación para comunicarte fácilmente entre todos tus subprogramas.
2. El segundo punto principal es que Ros2 te proporciona muchas herramientas y bibliotecas "plug and play" que te ahorrarán una gran cantidad de tiempo y, lo más importante, te evitarán reinventar la rueda.

Se dice que Ros2 es agnóstico al lenguaje, lo que significa que puedes programar algunas partes de tu aplicación en un lenguaje de programación y otra parte en otro lenguaje de programación simplemente porque las herramientas de comunicación no dependen de un lenguaje específico.

Terminal Terminator

Los atajos de teclado para dividir el terminal son "`Ctrl+Shift+O`" para dividir horizontalmente (arriba y abajo) y "`Ctrl+Shift+E`" para dividir verticalmente (izquierda y derecha).

Instalación de ROS2

La instalación de ROS2 no tiene mucho misterio, simplemente hay que seguir las instrucciones que aparecen en la siguiente web.

Instalación de Ignition gazebo

La instalación de ignition gazebo se hace siguiendo los pasos de esta página.
Los tutoriales para empezar a utilizar gazebo también se encuentran disponibles desde la misma web.

Para iniciar gazebo con interfaz gráfica: `gz sim` luego aparece un menú, pero entrando directamente desde él me da error y no entra. Para entrar bien hay que entrar desde consola con: `gz sim nombreSim.sdf`

Creando un paquete Python

Para crear un paquete Python desde la terminal vamos al directorio que queramos (de ahora en adelante siempre **/src**) y escribimos:

```
ros2 pkg create nombrePaquete --build-type ament_python
--dependencies nombreLibrería
```

Con esa línea creamos un paquete con el nombre que queramos, luego le decimos que será un paquete Python y le indicamos las dependencias necesarias (son opcionales, las puedes añadir luego). (rclpy)

Cada paquete de ROS2, ya sea en C++ o en Python, contendrá un archivo llamado "package.xml".

En ese archivo, básicamente encontrarás dos elementos:

1. El primero es una serie de información que incluye el nombre, la versión de tu paquete, una breve descripción y una lista de datos de mantenimiento seguida de la licencia. Esto está vacío por ahora. Si alguna vez deseas publicar tu paquete, compartirlo con la comunidad de código abierto o incluso agregar una licencia comercial, deberás editar estas etiquetas.

2. Luego, tenemos las dependencias que especificamos al crear el paquete. Si necesitas agregar una nueva dependencia, simplemente añadirás otra

etiqueta de dependencia debajo de esa. Justo después de las dependencias puedes ver el tipo de construcción en Python para el paquete.

Para compilar usando colcon desde el terminal simplemente escribimos `colcon build` (donde lo hayamos instalado), pero esto compila todo. Para compilar solo un paquete usamos: `colcon build --package-select nombrePaquete`

Creando un paquete C++

Crear un paquete C++ es pŕacticamente igual que si lo hacemos en Python, lo único que cambia es el ament, que utiliza cmake. El nombre y las librerías también depende de las necesidades. (rclcpp)

```
ros2 pkg create nombrePaquete --build-type ament_cmake
--dependencies nombreLibrería
```

La arquitectura del paquete de C++ es bastante diferente a la del paquete de Python, y eso se debe al cambio en el tipo de construcción.

En ese paquete de C++, tenemos un directorio "include" y uno "source". En C++, es común poner los archivos header en el directorio "include" y los archivos CPP en la carpeta "source". Ahora, tienes el archivo `package.xml`, que básicamente es lo mismo que en el paquete de Python: primero, la información y luego la biblioteca de C++.

Si necesitas otras dependencias, simplemente agregarás otra etiqueta `<depend>` debajo de esa. También hay un archivo ‘CMakeLists.txt` y cuando usamos C++, necesitamos compilar nuestro código, y aquí es donde lo harás. También indicarás dónde deseas instalar el código, etc.

Como puedes ver, también tenemos aquí la dependencia para C++. Si necesitas agregar una nueva dependencia para tu paquete de C++, deberás agregarla aquí en el `package.xml` y también en el `CMakeLists.txt`.

¿Qué es un nodo de ROS2?

Un nodo es una subparte de tu aplicación con un propósito único. Tu aplicación contendrá muchos nodos agrupados en paquetes, que se comunicarán entre sí.

Para entenderlo mejor, consideremos un ejemplo simplificado. Empezamos con un paquete de cámara, que manejará una cámara como una unidad independiente. Creamos nodos dentro del paquete, como un controlador para programar la cámara y obtener fotogramas, y un programa de procesamiento de imágenes.

Cada nodo puede lanzarse por separado. Los nodos se comunican mediante las funcionalidades de comunicación de ROS.

En otro ejemplo, tenemos un paquete de planificación de movimiento con nodos que calculan la planificación de movimiento y corrigen trayectorias según factores externos.

Un tercer paquete, control de hardware, controla el hardware del robot con nodos para controlar motores y publicar el estado del hardware.

Los nodos se comunican entre paquetes, vinculando nodos de procesamiento de imágenes a nodos de corrección de trayectoria, por ejemplo.

Los nodos reducen la complejidad del código al separar la aplicación en paquetes y proporcionan tolerancia a fallos al ejecutarse en procesos independientes. ROS2 es independiente del lenguaje, permitiendo nodos en Python y C++ que se comuniquen sin problemas.

Ten en cuenta que dos nodos no pueden tener el mismo nombre, así que en tus programas y en tu código, tendrás que asegurarte de que todos los nombres de los nodos sean únicos.

Nodos en C++

Para crear nuestro primer nodo, sigue estos pasos:

1. Crea un archivo .cpp en la carpeta "src" del paquete que hemos creado previamente. Por ejemplo, "**my_first_node.cpp**".
2. Incluye la biblioteca de ROS2 para C++ con `**#include "rclcpp/rclcpp.hpp"**`.

VS Code puede generar advertencias sobre la ruta de inclusión. Para solucionarlo, agrega la ruta de ROS al archivo `c_cpp_properties.json`.

```
"includePath": [
    "${workspaceFolder}/**",
    "/opt/ros/humble/include/**"
],
```

Ejemplo del Nodo Mínimo en C++:

```
#include "rclcpp/rclcpp.hpp"

int main(int argc, char **argv) {
    rclcpp::init(argc, argv); // Inicializa las comunicaciones
en ROS2
    auto node = std::make_shared<rclcpp::Node>("cpp_test"); //
Crea el nodo DENTRO del archivo, no es el archivo el nodo
    RCLCPP_INFO(node->get_logger(), "Hello cpp Node"); //
Imprime
    rclcpp::spin(node); // Mantiene el nodo activo y en
ejecución
    rclcpp::shutdown(); // Terminan las comunicaciones

    return 0;
}
```

Para compilar e instalar el nodo, configura el archivo `CMakeLists.txt` del mismo paquete.

```
# Crea el ejecutable e instala el nodo
add_executable(cpp_node src/my_first_node.cpp)
ament_target_dependencies(cpp_node rclcpp)

install(TARGETS
  cpp_node
  DESTINATION lib/${PROJECT_NAME}
)
```

Explicación:

- La primera línea crea el ejecutable. El primer argumento es el nombre del ejecutable, y el segundo es el archivo donde se encuentra el nodo.
- Se añaden las dependencias al ejecutable.

- Por último, se instala el nodo para poder ejecutarlo directamente con comandos de ROS2 en la consola.

Para ejecutarlo, usa el siguiente comando en la consola (se recomienda usar `source ~/.bashrc` antes para refrescar las configuraciones):

```
source ~/.bashrc
ros2 run nombre_paquete nombre_ejecutable
```

En este caso concreto:

```
ros2 run my_cpp_pkg cpp_node
```

En el primer paquete he dejado una plantilla para usarla y evitar tener que escribir el código repetitivo en cada archivo. Incluye una clase creada.

ROS2 Librerías para diferentes lenguajes

En primer lugar, es fundamental comprender la variedad de bibliotecas de cliente empleadas para interactuar con estas arquitecturas. Hasta ahora, hemos utilizado rclpy para Python y rlccpp para C++. ¿Pero de dónde provienen y por qué comparten el prefijo "rcl"?

"rcl" no es solo un prefijo; es una biblioteca de ROS2, que significa "ros client library". Esta biblioteca, escrita en C puro, aglutina las funcionalidades centrales bajo el nombre de rcl. Así que, en esencia, estas bibliotecas parten de la misma raíz.

Ahora, hablemos del middleware de ROS2 que se apoya en DDS (Servicio de Distribución de Datos). Este middleware gestiona todas las comunicaciones en la aplicación, proporcionando una capa esencial para el intercambio de información.

rcl, como biblioteca base para ROS sirve como puente hacia el middleware de ROS2. No obstante, en la práctica, no utilizamos rcl directamente. En su lugar, se opta por otras bibliotecas de cliente construidas sobre rcl.

Lo fascinante aquí es que, ya sea que programes en C++ o Python, estás trabajando con la misma biblioteca base, rcl. Esta coherencia es clave y simplifica el desarrollo en ambos lenguajes.

Además, aunque C++ y Python son los lenguajes más respaldados, la flexibilidad de la biblioteca permite la extensión a otros lenguajes. Ya existen implementaciones

para Node.js (rclnodejs) y Java (rcljava), lo que subraya la versatilidad de estas herramientas.

Monitorear nodos

Vamos a explorar la herramienta de línea de comandos "`ros2`". Cuando presionas la tecla Tab dos veces, se despliegan todos los comandos disponibles, y hay varios. Ya hemos visto "`ros2 run`" y "`ros2 package`" para la creación de paquetes; sin embargo, en este curso, conoceremos otros comandos.

Si escribes "`ros2 run`", este comando espera que le proporciones un paquete y un ejecutable. Esto te permite lanzar cualquier ejecutable desde la carpeta de instalación global de ROS2 o desde el espacio de trabajo de ROS2. Para obtener ayuda sobre el comando, puedes escribir "`ros2 run -h`" y recibirás un mensaje de ayuda, una práctica que puedes aplicar a cualquier otro comando para obtener documentación.

Otra herramienta útil es "`ros2 node`". Al escribir "`ros2 node`" y presionar Tab dos veces, obtienes una segunda acción. Por ejemplo, puedes utilizar "`ros2 node list`" para visualizar todos los nodos en ejecución en tu entorno. Esto facilita la supervisión de la actividad de los nodos.

Además, "`ros2 node info`" te proporciona información detallada sobre un nodo específico. Al ingresar el nombre del nodo, puedes obtener datos sobre sus subscriptores, publicadores, y más. Los publicadores y servidores de servicios que visualizas están presentes para todos los nodos. Por ejemplo, "rosout" actúa como un publicador general, recopilando registros de todas las aplicaciones. También hay publicadores y servidores de servicios destinados a los parámetros, y cada nodo gestiona sus propios parámetros.

En resumen, hemos explorado el comando "`ros2 pkg create`" para la creación de un nuevo paquete. También hemos visto cómo utilizar "`ros2 run`" con el nombre del paquete y el ejecutable asociado. Además, los comandos "`ros2 node list`" y "`ros2 node info`" permiten listar nodos y obtener información detallada sobre ellos, respectivamente.

Y es muy importante notar que no deberías tener dos nodos con el mismo nombre en tu gráfico.

Renombrar nodos en tiempo de ejecución

En tu aplicación de ROS2, es posible que desees iniciar el mismo nodo varias veces, pero con una configuración diferente.

Supongamos que cuentas con un nodo para un sensor de temperatura y necesitas manejar cinco sensores en total. En este caso, tendrás que lanzar el mismo nodo cinco veces, asignando un nombre único a cada sensor para su identificación.

Es crucial tener en cuenta que no puedes lanzar el mismo nodo con el mismo nombre más de una vez. Aunque no recibiras un error al hacerlo, veamos las posibles consecuencias.

Al utilizar el comando "`ros2 node list`" para listar los nodos, surge una advertencia indicando que algunos nodos comparten un nombre idéntico en el gráfico, lo cual podría ocasionar efectos secundarios no deseados. Al revisar la información de un nodo específico mediante "`ros2 node info`", también se emite una advertencia sobre la existencia de dos nodos con el mismo nombre, y solo se muestra información acerca de uno de ellos.

A diferencia de ROS1, en ROS2 ahora es posible iniciar dos nodos con el mismo nombre. Sin embargo, aunque sea técnicamente posible, no se aconseja hacerlo debido a los posibles problemas que podría generar en el gráfico y las comunicaciones.

Entonces, ¿cómo puedes abordar esta situación cuando necesitas iniciar el mismo nodo múltiples veces, como en el caso del sensor de temperatura? La solución es sencilla: simplemente debes cambiar el nombre del nodo al iniciarlo.

Para lograr esto, puedes emplear el comando "`ros2 run`" y agregar argumentos específicos después de "`--ros-args`". Un ejemplo práctico sería utilizar "`--remap`" seguido de "`__node:=miNodo`" para asignar el nombre "`miNodo`" al nodo. Es esencial destacar que este nombre no está fijado de manera permanente y puede modificarse dinámicamente al iniciar el nodo con "`ros2 run`".

Esta característica resulta especialmente útil cuando necesitas duplicar cierto comportamiento y deseas contar con nodos que tengan configuraciones únicas sin tener que realizar modificaciones en el código en cada ocasión.

Ejemplo:

```
ros2 run my_cpp_pkg cpp_node --ros-args --remap __node:=miNodo
ros2 run my_cpp_pkg cpp_node --ros-args -r __node:=miNodo
```

Rqt y rq_graph

Veamos cómo funciona rqt, una herramienta para resolver problemas en tu gráfico y nodos de ROS2.

Para empezar, simplemente escribe "`rqt`" en la consola para abrir esta herramienta de depuración. Aunque podrías usar "`ros2 run`", la buena noticia es que rqt tiene su propio ejecutable, haciéndolo más fácil de usar.

Cuando entras a rqt, te enfrentas a una pantalla vacía, pero no te preocupes, es normal. rqt es como una caja de herramientas, y el elemento que nos interesa aquí es el "gráfico de nodos" (node graph).

El "gráfico activo" muestra el estado actual del gráfico. Como aún no hemos iniciado ningún nodo, está vacío. Vamos a probar iniciando un nodo para ver qué pasa. Al actualizar la pantalla en rqt, verás el nodo recién iniciado. Si lanzamos el mismo nodo con otro nombre y actualizamos la pantalla, ahora vemos dos nodos.

Es clave tener en cuenta que la ventana de rqt en sí misma es un nodo, como se refleja en la lista de nodos. Si quitas la opción de depuración y desactivas "dead sink", podrás observar el tópico "rosout" al cual cada nodo publica. Aunque los nodos operan de manera independiente, todos se comunican mediante este tópico.

Cuando generas registros (logs), estos se envían tanto a "rosout" como a la interfaz de línea de comandos de ROS2 (ROS2 CLI). Además, hay un "daemon" que se inicia al arrancar un nodo, facilitando la interconexión entre nodos, aunque por ahora no es necesario preocuparse por ello.

También puedes arrancar directamente el gráfico con el ejecutable "rqt_graph", una opción más rápida pero equivalente. El uso de rqt graph resulta muy útil para obtener una visión global de lo que sucede en tu gráfico de ROS2 y para identificar fácilmente posibles errores.

Primer contacto con turtle-sim

Vamos a experimentar con un paquete ya existente que proporciona una simulación simplificada de un robot. Este paquete se llama "turtlesim".

Si no tienes el paquete de turtlesim instalado, puedes hacerlo escribiendo simplemente `` `sudo apt install ros-distro-turtlesim` ``, donde "distro" es el nombre de la distribución que estás utilizando (por ejemplo, "humble").

Una vez instalado, puedes abrir una nueva terminal para depurar, o simplemente actualizar la terminal actual usando `` `source ~/.bashrc` ``, ya que el paquete está instalado, pero la terminal aún no lo reconoce.

Luego, puedes ejecutar el comando `` `ros2 run turtlesim turtlesim_node` `` en la terminal para iniciar un nodo que abrirá una ventana con una tortuga en el medio.

Un nodo puede hacer mucho más que simplemente mostrar una salida en la terminal. Puedes realizar visualizaciones, trabajar en red y mucho más. Después, puedes cerrar la comunicación del nodo cuando hayas terminado.

Si ejecutas `` `ros2 node list` ``, verás el nodo llamado "/turtlesim". Ahora, cambiemos el diseño y arranquemos otro nodo para mover la tortuga, para ello escribimos: ‘`ros2 run turtlesim turtle_teleop_key`’. Puedes utilizar las teclas de flecha para mover la tortuga en la pantalla. Este nodo recibe tus inputs, los envía al nodo "/turtlesim" y muestra el resultado en la pantalla.

Si ejecutas `rqt_graph` y actualizas, podrás ver la entrada de los nodos "/turtlesim" y “/teleop_turtle” y cómo están comunicándose entre sí.

Topics

Los "topics" (temas) de ROS2, son un concepto que se puede entender mejor a través de una analogía con un transmisor y receptor de radio.

Imaginemos un transmisor de radio como un nodo que publica datos en un tema específico, representado por un número o frecuencia, por ejemplo, 98.7. Este número actúa como un identificador, permitiendo que los nodos receptores, como un teléfono o un automóvil, se suscriban a ese tema para recibir la información transmitida.

En esta analogía, el transmisor de radio se convierte en un editor (publisher) que envía datos sobre el tema 98.7, mientras que los dispositivos receptores, como teléfonos o automóviles, actúan como suscriptores (subscribers) que reciben y decodifican la información. Es crucial destacar que tanto los editores como los suscriptores deben utilizar la misma estructura de datos (interface) para asegurar una comunicación efectiva.

La flexibilidad de este sistema se revela cuando se consideran múltiples nodos y su capacidad para publicar en y suscribirse a varios temas. Un nodo puede actuar como un editor en un tema específico, como el transmisor de radio que publica tanto señales FM como AM en diferentes temas, y simultáneamente puede ser un suscriptor en otro tema, como un automóvil que recibe datos de ubicación mientras escucha la radio.

La independencia de los nodos es un aspecto fundamental. Cada nodo, ya sea un editor o un suscriptor, opera de manera autónoma, sin conocimiento de los demás participantes en el tema. Esto permite diversas combinaciones, como nodos que solo publican, solo suscriben o realizan ambas funciones.

El uso de temas se destaca en situaciones donde se requiere el envío de flujos de datos unidireccionales. Los temas proporcionan un canal de comunicación efectivo donde los editores publican información y los suscriptores la reciben, sin la necesidad de respuestas desde el suscriptor hasta el editor. Además, tanto los editores como los suscriptores son anónimos, enfocándose únicamente en la interacción a través del tema en cuestión.

En términos prácticos, la documentación de ROS2 define un tema como un bus nombrado a través del cual los nodos intercambian mensajes. Es esencial notar que, aunque en la analogía se usaron números con puntos como nombres de temas, en la implementación real, un nombre de tema debe cumplir con ciertos requisitos, como comenzar con una letra y permitir letras, números, guiones bajos, guiones y barras inclinadas.

Consideraciones sobre los “topics”

- ➢ El flujo de datos es unidireccional, por lo que los nodos pueden publicar en el tema y algunos nodos pueden suscribirse al tema.

- ➢ No hay respuesta de un suscriptor a un editor.

- ➢ Los datos van en una sola dirección.

- ➢ Los editores y suscriptores son anónimos.
- ➢ Un editor solo sabe que está publicando en un tema y un suscriptor solo sabe que está suscribiéndose a un tema, nada más.
- ➢ Un tema tiene un tipo de mensaje.
- ➢ Todos los editores y suscriptores en el tema deben usar el mismo tipo de mensaje asociado con el tema.

Publishers en C++

Un publicador (publisher) es un nodo que crea y envía mensajes a un tema específico. Los suscriptores que escuchan en ese tema pueden recibir y procesar estos mensajes. Aquí hay algunos detalles sobre los publicadores en ROS2:

1. Creación de un publicador: En C++, un publicador se crea utilizando el método `create_publisher` del nodo. Necesitas especificar el tipo de mensaje que el publicador va a enviar y el nombre del tema al que va a publicar. También puedes especificar el tamaño de la cola de mensajes para el publicador.

```
publisher_ =
this->create_publisher<example_interfaces::msg::String>("topic
_name", 10);
```

2. Publicación de mensajes: Una vez que tienes un publicador, puedes usarlo para publicar mensajes. Primero, creas un mensaje del tipo apropiado, luego lo llenas con datos y finalmente lo publicas con el método `publish` del publicador.

```
auto message = example_interfaces::msg::String();
message.data = "Hello, world!";
publisher_->publish(message);
```

3. Ciclo de vida del publicador: El publicador permanece activo y puede publicar mensajes mientras el nodo que lo creó esté activo. Cuando el nodo se destruye, también se destruye el publicador.

4. Punteros compartidos: En C++, los publicadores se manejan a menudo como punteros compartidos (`std::shared_ptr`). Esto significa que el publicador se destruirá automáticamente cuando ya no haya referencias a él.

5. QoS (Quality of Service): Al crear un publicador, puedes especificar una política de QoS que controla cómo se entregan los mensajes. Esto puede incluir cosas como la confiabilidad de la entrega de mensajes, la duración de los mensajes y la política de historia.

Subscribers en C++

En ROS2, un suscriptor (subscriber) es un nodo que se suscribe a un tema específico para recibir y procesar mensajes enviados a ese tema por los publicadores. Aquí hay algunos detalles sobre los suscriptores en ROS2:

1. Creación de un suscriptor: En C++, un suscriptor se crea utilizando el método `create_subscription` del nodo. Necesitas especificar el tipo de mensaje que el suscriptor va a recibir y el nombre del tema al que se va a suscribir. También debes proporcionar una función de devolución de llamada que se invocará cada vez que se reciba un mensaje. Además, puedes especificar el tamaño de la cola de mensajes para el suscriptor.

```
subscriber_ =
this->create_subscription<example_interfaces::msg::String>("to
pic_name", 10, std::bind(&ClassName::callbackFunction, this,
std::placeholders::_1));
```

2. Función de devolución de llamada: La función de devolución de llamada es una función que se invoca cada vez que se recibe un mensaje. Esta función recibe el mensaje como argumento y puede procesarlo como sea necesario.

```
void callbackFunction(const
example_interfaces::msg::String::SharedPtr msg){
       // Procesar el mensaje
}
```

3. Ciclo de vida del suscriptor: El suscriptor permanece activo y puede recibir mensajes mientras el nodo que lo creó esté activo. Cuando el nodo se destruye, también se destruye el suscriptor.

4. Punteros compartidos: En C++, los suscriptores se manejan a menudo como punteros compartidos (`std::shared_ptr`). Esto significa que el suscriptor se destruirá automáticamente cuando ya no haya referencias a él.

5. QoS (Quality of Service): Al crear un suscriptor, puedes especificar una política de QoS que controla cómo se reciben los mensajes. Esto puede incluir cosas como la confiabilidad de la entrega de mensajes, la duración de los mensajes y la política de historia.

Debug ROS2 Topics usando CLI

Si deseamos obtener más información sobre un tema, escribiremos `'ros2 topic info'` seguido del nombre del tema. Este comando nos proporciona detalles cruciales, como el tipo de datos manejado, así como el número de publicadores y suscriptores asociados. Es útil para comprender la dinámica de intercambio de información en nuestro sistema.

'`ros2 topic echo`' permite "escuchar" los mensajes en tiempo real. Además, con '`ros2 interface show`' se puede visualizar la definición precisa del mensaje, aportando claridad sobre los datos que debemos enviar al tema.

Con '`ros2 topic hz /nombre_topic`', podemos obtener la frecuencia de publicación real del topic.

'`ros2 topic bw /nombre_topic`', nos da información sobre el ancho de banda utilizado por el nodo. Esto puede ser crucial para optimizar el rendimiento, especialmente cuando lidiamos con tasas de publicación elevadas.
Se puede publicar directamente desde la terminal con '`ros2 topic pub`'. Aunque este comando puede no usarse a menudo, es valioso conocerlo para situaciones específicas.

Remap topics en tiempo de ejecución

Es exactamente igual que renombrar los nodos, solo hay que escribir lo siguiente:

```
ros2 run my_cpp_pkg cpp_node --ros-args -r
nombreTopic:=NuevoNombre
```

Si quieres renombrar el nodo y el topic lo puedes hacer en la misma instrucción:

```
ros2 run my_cpp_pkg cpp_node --ros-args -r __node:=miNodo -r
nombreTopic:=NuevoNombre
```

Debemos tener en cuenta que, por ejemplo, si un publisher renombra el topic, el suscriptor debe usar el nuevo nombre del topic, de lo contrario no recibirá nada.

Servicios

Uno de los conceptos fundamentales es el servicio ROS2. Para comprender mejor este componente central, podemos recurrir a analogías de la vida cotidiana. Imaginemos un servicio meteorológico en línea como ejemplo representativo.

En esta analogía, el usuario, o cliente, busca obtener información meteorológica localizada a través de una plataforma en línea. En términos de ROS2, el cliente sería nuestro propio ordenador, mientras que el servicio meteorológico en línea actuaría como el servidor. La conexión entre ambos se establece mediante solicitudes HTTP y una URL, que en la jerga de ROS2 equivaldría al nombre del servicio.

Para simplificar, la interacción se inicia con el cliente enviando una solicitud al servidor, llevando consigo un mensaje que contiene información crucial, como la ubicación para obtener datos meteorológicos específicos. El servidor procesa la solicitud y devuelve una respuesta con los datos solicitados. Es vital destacar que para que esta comunicación sea efectiva, el mensaje enviado por el cliente debe ser estructurado de manera que el servidor pueda procesarlo adecuadamente. La respuesta del servidor, a su vez, también lleva un mensaje para asegurar una comprensión mutua.

En este punto, surge la pregunta: ¿Qué sucede cuando hay múltiples clientes? La respuesta es que todos los clientes pueden enviar solicitudes al servidor de manera simultánea, cada uno con su propia ubicación y recibiendo respuestas personalizadas. No obstante, es crucial mantener un único servidor para evitar complicaciones.

En el contexto de ROS2, los nodos de computadora representan entidades independientes que actúan como clientes, cada uno con su propio servicio cliente. Por otro lado, el servidor de servicios ROS2 procesa las solicitudes y devuelve respuestas a cada cliente, manteniendo la simplicidad y la independencia entre ellos.

Al trasladarnos a un ejemplo más directamente vinculado con la robótica, consideremos un nodo que controla un panel LED y otro que gestiona la batería de un robot. Aquí, se introduce un servicio ROS2 llamado "set" que permite encender o apagar LEDs específicos. El nodo del panel actúa como servidor, procesando solicitudes de encendido o apagado provenientes de nodos clientes, como el nodo

de la batería. Este último puede enviar solicitudes al servidor para, por ejemplo, apagar un LED cuando la batería está baja. La comunicación entre estos nodos, facilitada por el servicio ROS2, garantiza una interacción efectiva y específica.

Características:

- ➢ Un servicio de ROS2 es un sistema cliente-servidor.
- ➢ Puede ser síncrono o asíncrono.
- ➢ Un servicio se define como un nombre y un par de mensajes. Un mensaje es la solicitud y el otro es la respuesta.
- ➢ Al igual que con nodos y temas, puedes crear directamente clientes y servidores de servicio dentro de nodos de ROS2 utilizando C++ o Python
- ➢ Un servidor de servicio solo puede existir una vez, pero puede tener muchos clientes y básicamente el servicio comenzará a existir cuando creas el servidor.

Por lo tanto, los servicios de ROS2 se revelan como sistemas cliente-servidor, poniendo de manifiesto su utilidad en escenarios donde se requiere una arquitectura de comunicación más estructurada y específica que la proporcionada por los temas de ROS2. Así, mientras que los temas se centran en flujos de datos unidireccionales, los servicios emergen como la herramienta ideal cuando se busca una interacción más detallada y controlada entre nodos en un entorno robótico.

Debug de los servicios usando rqt

Escribiendo ‘rqt’ en el terminal nos aparecerá una ventana de herramientas. Desplegando la pestaña ‘plugins’ y seleccionando ‘services’ -> ‘service caller’ se abrirá una interfaz gráfica para probar los servicios en funcionamiento.

Remap servicios en runtime

Lo mismo que para nodos y topics, pero haciendo referencia a el servicio que queremos renombrar.

Ejemplo usando el archivo add_two_ints_server.cpp

```
ros2 run my_cpp_pkg add_two_ints_server --ros-args -r
add_two_ints:=NuevoNombre
```

Interfaces en ROS2

Para entender la noción de interfaz en ROS2, es crucial abordar primero los conceptos fundamentales de "temas" y "servicios". En las secciones previas, hemos experimentado cómo estas entidades actúan como mediadores para la transmisión de información entre nodos en un sistema ROS2.

En términos simples, un tema en ROS2 está definido por dos elementos esenciales: un nombre distintivo y una interfaz basada en la definición de un mensaje. Este mensaje, a su vez, determina la estructura de los datos que se intercambian entre los nodos. Por otro lado, un servicio en ROS2 también se caracteriza por un nombre y una interfaz, pero esta interfaz comprende dos mensajes: uno para la solicitud y otro para la respuesta.

Para visualizar este proceso, podemos recurrir a una analogía simple. Imaginemos que enviar un mensaje entre nodos es como enviar una carta por correo convencional. En este escenario, la empresa de transporte representa la capa de comunicación (temas y servicios), mientras que el contenido de la carta se asemeja a una interfaz ROS2, utilizando una definición de mensaje específica.

Al enviar una carta y esperar una respuesta, estamos esencialmente siguiendo el patrón de un servicio en ROS2, donde la carta original contiene un mensaje de solicitud y la respuesta recibida incluye un mensaje de respuesta. La combinación de las definiciones de estos dos mensajes constituye la definición de servicio.

En la infraestructura de ROS2, los temas y servicios actúan como la capa de comunicación, mientras que las herramientas y las interfaces se asemejan a los mensajes reales que se envían. Esto nos lleva a la pregunta crítica: ¿cómo podemos utilizar estas interfaces directamente en nuestro código?

La respuesta radica en la creación de definiciones de mensajes dentro de un paquete. Cuando utilizamos el comando de construcción correspondiente, el sistema de construcción de ROS2 genera el código fuente necesario en diversos lenguajes compatibles, como C++ o Python. Este código resultante permite la inclusión directa de las definiciones de mensaje en nuestro código, facilitando así la implementación de interfaces en nuestras aplicaciones.

Otro aspecto crucial de las interfaces en ROS2 es la variedad de tipos de datos que podemos emplear en la creación de definiciones de mensajes y servicios. Desde booleanos hasta números de punto flotante, enteros y cadenas, ROS2 ofrece una gama versátil de tipos de datos primitivos. Además, podemos utilizar matrices de estos tipos primitivos, ampliando aún más nuestras opciones.

Para obtener una referencia exhaustiva de estos tipos de datos, la página de interfaces de ROS2 en el sitio web del índice de ROS proporciona una guía detallada. Allí, encontramos información sobre cómo estos tipos de datos se traducen a lenguajes de programación específicos, como C++ y Python.

Para ilustrar estos conceptos, examinemos el paquete de interfaces de ejemplo disponible en GitHub bajo la organización de ROS2. Este paquete no solo sirve como un conjunto de ejemplos, sino que también nos muestra cómo las definiciones de mensajes y servicios se materializan en la práctica.

Dentro del paquete, encontramos mensajes que abordan una variedad de situaciones, desde mensajes simples como INT64 hasta definiciones de servicios más complejas como set bool. Cada uno de estos ejemplos nos permite visualizar cómo se estructuran y utilizan las interfaces en un contexto real.

Además de los ejemplos básicos, es esencial explorar las interfaces más complejas que la comunidad de ROS2 ha desarrollado. Algunos paquetes, como el de mensajes de sensores, contienen definiciones detalladas para situaciones específicas. Por ejemplo, podemos observar la definición de un mensaje que permite el envío de información detallada sobre la posición, velocidad y esfuerzo de cada articulación de un robot.

En última instancia, las interfaces en ROS2 desempeñan un papel central en el desarrollo de aplicaciones robóticas y de automatización. Son los bloques de construcción que facilitan la comunicación y la colaboración entre nodos, permitiendo la creación de sistemas robóticos complejos y altamente eficientes.

Para incorporar estas interfaces en nuestro propio código, el proceso se simplifica mediante la creación de definiciones de mensajes y servicios personalizados dentro de nuestros paquetes. La generación de código resultante a través del sistema de construcción de ROS2 nos brinda la capacidad de integrar directamente estas interfaces en nuestras aplicaciones, haciendo que la comunicación entre nodos sea más accesible y efectiva.

Configurar pkg para las interfaces personalizadas

Lo primero que debemos abordar es dónde escribiremos nuestro mensaje personalizado. En ROS2, la organización en paquetes es crucial, y aunque técnicamente podemos agregar definiciones de mensajes en cualquier paquete, es altamente recomendable crear un paquete dedicado exclusivamente a nuestras interfaces personalizadas. Este enfoque no solo simplificará la gestión de dependencias, sino que también evitará confusiones y problemas en el futuro.

En el mundo de ROS2, los paquetes como "sensor messages" y "geometry messages" son ejemplos de paquetes dedicados exclusivamente a mensajes, sin incluir código. Siguiendo esta lógica, al crear un paquete específico para nuestras interfaces, garantizamos una clara separación entre la capa de comunicación y la implementación del código.

Para empezar, crearemos un nuevo paquete en nuestro espacio de trabajo ROS2 utilizando el comando `ros2 create package`. Aquí, un nombre sugerido podría ser "**my_robot_interfaces.**" No es necesario especificar un tipo de construcción ni agregar dependencias, ya que este paquete será exclusivamente para nuestras interfaces.

Ahora, dentro de nuestro nuevo paquete, eliminaremos las carpetas "include" y "source," ya que no contendrá código. En su lugar, crearemos una carpeta "**msg**" para guardar nuestras definiciones de mensajes personalizados.

Para configurar el paquete primero debemos añadir estas tres líneas en package.xml, dentro de <package></package>

```
<build_depend>rosidl_default_generators</build_depend>
<exec_depend>rosidl_default_runtime</exec_depend>
<member_of_group>rosidl_interface_packages</member_of_group>
```

El siguiente paso es la creación de nuestro primer mensaje personalizado. Por ejemplo, imaginemos que queremos enviar información sobre el estado del hardware de nuestro robot. Creamos un archivo llamado "HardwareStatus.msg" en la carpeta "msg" y definimos los campos, como la temperatura del sensor, la disponibilidad de motores y un mensaje de depuración. Por convención el nombre de las interfaces se escribe en PascalCase.

Así queda “HardwareStatus.msg”

```
int64 temperature
```

```
bool are_motors_ready
string debug_message
```

Para terminar la configuración, el “CMakeList.txt” quedaría tal que así, simplemente a cada msg que creemos habría que añadirlo donde corresponde. Las líneas en negrita son las nuevas que debemos añadir.

```
cmake_minimum_required(VERSION 3.8)
project(my_robot_interfaces)

if(CMAKE_COMPILER_IS_GNUCXX OR CMAKE_CXX_COMPILER_ID MATCHES
"Clang")
  add_compile_options(-Wall -Wextra -Wpedantic)
endif()

# find dependencies
find_package(ament_cmake REQUIRED)
find_package(rosidl_default_generators REQUIRED)

rosidl_generate_interfaces(${PROJECT_NAME}
  "msg/HardwareStatus.msg"
  #Añadir msg nuevos aquí
)

ament_export_dependencies(rosidl_default_runtime)

ament_package()
```

Una vez hecho todo, solo hay que compilar el paquete como hemos hecho hasta ahora y la interfaz estará lista para su uso.

Usar las interfaces personalizadas en C++

Para utilizar el mensaje creado anteriormente, debemos incluir lo siguiente en los archivos que lo vayan a utilizar:

```
#include “my_robot_interfaces/message/hardware_status.hpp”
```

En VS Code no encontrará el header, eso es porque debemos ir al archivo de propiedades “**c_cpp_properties.json”** e incluir la ruta:

```
"includePath": [
```

```
    "${workspaceFolder}/**",
    "/opt/ros/humble/include/**",
    "~/ros2_ws/install/my_robot_interfaces/include/**"
],
```

No olvidar incluir en el “**package.xml**” la dependencia:

```
<depend>my_robot_interfaces</depend>
```

Y en “**CMakeList.txt**”:

```
find_package(my_robot_interfaces REQUIRED)
```

Con esto ya estaría todo configurado para que encuentre el nuevo paquete. Ahora se puede utilizar la nueva interfaz igual que usábamos las de ejemplo ya creadas.

Crear srv personalizados

En la creación de interfaces personalizadas se usaba como ejemplo un mensaje, para los servicios es exactamente lo mismo solo que tenemos que crear una nueva carpeta que se llame “**srv**” para que esté todo ordenado y acabar el nombre de los archivos con “**.srv**”. Un ejemplo de cómo sería un servicio es el siguiente:

“ComputeRectangleArea.srv”

```
float64 length
float64 width
---                 # Separan las peticiones (arriba) de las respuestas(abajo)
float64 area
```

Parámetros

Para comprender a fondo los parámetros, es necesario comenzar examinando el problema que resuelven.

Para ilustrar este concepto, regresemos al paquete de la cámara, previamente utilizado para explicar los nodos y paquetes. Dentro de este paquete, encontramos un nodo controlador de cámara, y es precisamente en este nodo donde dirigimos nuestra atención.

¿Qué función cumple exactamente este nodo controlador de cámara? Su propósito fundamental radica en establecer una conexión con la cámara a través de una interfaz de hardware, como USB, para luego adquirir las imágenes generadas por dicha cámara.

Dentro de este nodo, nos topamos con variables que configuran su comportamiento. Por ejemplo, tenemos la opción de especificar el nombre del dispositivo USB al cual deseamos conectarnos y decidir si queremos que el nodo se ejecute en un modo de simulación. Naturalmente, podríamos tener numerosas configuraciones adicionales, lo que hace evidente que codificar estas opciones directamente en el código no es una opción práctica.

Es aquí donde los parámetros de ROS 2 emergen como una solución elegante. Un parámetro de ROS 2 es, esencialmente, un valor de configuración para un nodo. Este concepto resulta invaluable para cualquier tipo de configuración, ya que permite declarar estos parámetros en el código y, al iniciar el nodo, asignarles valores específicos.

Tomemos, por ejemplo, el caso de iniciar el nodo de la cámara y proporcionar valores para los parámetros. Estos valores se integran dinámicamente en el código, permitiendo al nodo adaptarse a distintas condiciones sin necesidad de intervención directa en su implementación.

Esta flexibilidad se traduce en una serie de beneficios significativos. Si, por ejemplo, deseamos ejecutar la cámara con un dispositivo USB diferente, simplemente ajustamos ese parámetro sin necesidad de recompilar el código. ¿Y si queremos iniciar la cámara con una velocidad de 30 fps en lugar de 60 fps? Nuevamente, modificamos el parámetro correspondiente sin incurrir en el tedioso proceso de recompilación.

La verdadera joya de los parámetros de ROS 2 radica en su capacidad para permitir la ejecución de nodos con diferentes configuraciones sin modificar el código fuente. Si queremos ejecutar el nodo en modo de simulación para una cámara y luego en modo real para otra, la transición es tan simple como ajustar los parámetros antes de iniciar el nodo.

Por lo tanto, los parámetros de ROS2 son esenciales para proporcionar configuraciones dinámicas a los nodos. La clave reside en establecer y ajustar estos valores en tiempo de ejecución, evitando así la necesidad de recompilar el código.

Cabe destacar que cada parámetro es específico de un nodo y persiste solo mientras el nodo esté activo. Dos nodos diferentes tendrán conjuntos distintos de valores para sus respectivos parámetros.

Un parámetro de ROS2 se define por su nombre y tipo de datos, siendo los tipos más comunes booleanos, enteros, decimales, cadenas y listas de estos tipos. Esta versatilidad facilita la adaptación de los nodos a una amplia gama de configuraciones sin comprometer la eficiencia del desarrollo.

Declarar los parámetros

En el terminal, "`ros2 param list`" mostrará una lista de los parámetros de cada nodo activo. Cada nodo tiene una lista de parámetros asociados. Por defecto cada nodo tiene un parámetro llamado "**use_sim_time**" establecido automáticamente por ROS2.

"`ros2 parameter get`" se utiliza para obtener un parámetro específico y su valor.

Aunque dos nodos puedan tener el mismo nombre de parámetro, los valores son específicos para cada nodo. Cambiar el valor de un nodo no afectará al otro. Cada parámetro existe en el ámbito del nodo y se pierde al eliminar el nodo.

Para declarar un parámetro en C++ dentro del nodo usaremos el puntero oculto "**this**". (se puede hacer directamente sin el '**this**')

```
this->declare_parameter<int>("nombre" , 3);
```

Si lo declaras con un valor, infiere el tipo a partir de ese valor. Si no le quieres poner valor de inicio le tienes que especificar el tipo obligatoriamente.

Una vez declarado el parámetro, puedes usarlo en variables o como argumento de funciones, por ejemplo:

```
int number_{};
number_ = this->get_parameter("number_to_publish").as_int();
```

`as_int()` convierte el valor del parámetro a un entero. Si el valor del parámetro no puede ser convertido a un entero, se lanzará una excepción. Dependiendo del tipo que sea la variable habrá que usar un tipo u otro, por ejemplo si '**number_**' fuera double se usaría `.as_double()`.

Para iniciar un nodo de ros2 desde consola con varios parámetros:

```
ros2 run my_cpp_pkg cpp_node —ros-args -p
nombreParametro:=valor -p nombreParametro:=valor
```

Launch files

En el desarrollo de aplicaciones robóticas, especialmente aquellas que involucran múltiples nodos y numerosos parámetros, la configuración manual de cada componente puede convertirse en una tarea laboriosa y propensa a errores. Tomemos, por ejemplo, el caso de un nodo controlador de cámara en el que se deben especificar parámetros como el nombre del dispositivo USB, las velocidades y una bandera de modo de simulación.

A medida que se añaden más componentes al robot, como cámaras adicionales o estaciones nuevas, la complejidad del despliegue se incrementa exponencialmente. La necesidad de iniciar cada nodo en una terminal diferente con configuraciones específicas se convierte en una tarea monumental, consumiendo tiempo y aumentando la posibilidad de errores.

Es en este punto donde los archivos de lanzamiento en Ros2 juegan un papel crucial. Estos archivos permiten consolidar todos los nodos y configuraciones en un solo archivo, simplificando enormemente el proceso de inicio. En lugar de iniciar manualmente cada nodo con sus parámetros, un simple script puede ser ejecutado para automatizar todo el proceso.

Beneficios de los Launch Files:

1. **Eficiencia en el Despliegue:** La capacidad de iniciar todos los componentes de la aplicación desde un solo archivo proporciona una eficiencia notable. Esto es especialmente valioso en entornos donde el tiempo de despliegue es crítico.

2. **Escalabilidad**: Los archivos de lanzamiento permiten escalar fácilmente una aplicación robótica. A medida que se añaden más nodos y parámetros, la complejidad no se traduce en complicaciones en el proceso de inicio.

3. **Facilidad de Debugging:** La consolidación de la configuración en un archivo único facilita la identificación y corrección de errores. En lugar de buscar configuraciones dispersas en diferentes terminales, todo el contexto está disponible en un solo lugar.

Crear e instalar Launch Files

Al igual que las interfaces, los launch files tendrán su propio paquete, ya que será más fácil si todos tus archivos de lanzamiento están dentro de un solo paquete. Los archivos de lanzamiento recogerán algunos nodos de muchos paquetes diferentes, por lo que será más fácil centralizarlos en uno solo. Además, reducirá el número de dependencias entre tus paquetes.

Creamos el paquete para los launchers como lo hemos hecho en los anteriores paquetes, sin dependencias ya que no serán necesarias.

```
ros2 pkg create my_robot_bringup
```

¿Y por qué llamamos al paquete `my_robot_bringup`? Esto no es aleatorio. Es algo que se utiliza con frecuencia en la comunidad de ROS. Colocas el nombre de tu robot y luego "bringup". Si adoptas esta convención, te resultará más fácil trabajar con el código de otras personas, y a otros les resultará más fácil trabajar con tu código.

Una vez creado, podemos borrar las carpetas de include y src y creamos otra con el nombre de launch, dónde irán los launch files.

En ‘package.xml’ hay que añadir las dependencias de ejecución de los paquetes donde se encuentren los nodos que utilicemos en los launcher tal que así:

```
<exec_depend>my_cpp_pkg</exec_depend>
```

El `CMakeLists.txt`, quedaría como sigue:

```
cmake_minimum_required(VERSION 3.8)
project(my_robot_bringup)

if(CMAKE_COMPILER_IS_GNUCXX OR CMAKE_CXX_COMPILER_ID MATCHES
"Clang")
  add_compile_options(-Wall -Wextra -Wpedantic)
endif()

# find dependencies
find_package(ament_cmake REQUIRED)

install(DIRECTORY
  launch
  DESTINATION share/${PROJECT_NAME}
```

```
)

ament_package()
```

Una vez configurado todo, creamos nuestro primer launch file dentro de la carpeta `launch`. Los archivos de lanzamiento son **archivos Python**, por lo que añadimos la extensión `.py`. Un ejemplo sería:

`counter_app.launch.py`

Así quedaría un archivo launch con la descripción de lo que hace el código:

```
from launch import LaunchDescription
from launch_ros.actions import Node

def generate_launch_description():
    ld = LaunchDescription()

    number_publisher_node = Node(
        package="my_cpp_pkg",
        executable="number_publisher"
    )

    number_counter_node = Node(
        package="my_cpp_pkg",
        executable="number_counter"
    )

    ld.add_action(number_publisher_node)
    ld.add_action(number_counter_node)

    return ld
```

1. Importa las clases necesarias: `LaunchDescription` y `Node` de los módulos `launch` y `launch_ros.actions` respectivamente.

2. Define una función `generate_launch_description()`. Esta función es requerida por ROS2 y será llamada cuando se ejecute el script de lanzamiento.

3. Dentro de la función, crea una instancia de `LaunchDescription`. Esta instancia actúa como un contenedor para las acciones que se realizarán durante el lanzamiento, como iniciar nodos.

4. Crea dos nodos utilizando la clase `Node`. El primer nodo es `number_publisher` del paquete `my_cpp_pkg` y el segundo nodo es `number_counter` del mismo paquete.

5. Añade ambos nodos a la descripción del lanzamiento utilizando el método `add_action`.

6. Finalmente, la función devuelve la descripción del lanzamiento.

Comentar que en los archivos de lanzamiento de ROS2 en Python, la ruta al ejecutable debe ser relativa al paquete en el que se encuentra. No se debe especificar la ruta absoluta al archivo.

Para comprobar que funciona hay que compilar el paquete y luego escribir en el terminal:

```
ros2 launch my_robot_bringup number_app.launch.py
```

Cuando se ejecute este script de lanzamiento, ROS2 iniciará los nodos `number_publisher` y `number_counter` del paquete `my_cpp_pkg`.

Configurar los nodos en el launch file

Podemos remapear el nombre de un nodo usando `name=""`, remapear topics y servicios usando `remapping=[ ("NombreViejo", "NuevoNombre" ) ]` o configurar parámetros con `parameters=[ {"NombreParametro": valor} ]`

Notar que los topics y servicios van en tuplas entre paréntesis ( ) y los parámetros van entre llaves { } y, después del nombre del parámetro, se usan los dos puntos antes del valor `:`

Es posible usar variables como "alias" para hacerlo más legible y mantenible.

Un ejemplo usando parte del código de la sección anterior quedaría como sigue:

```
remap_number_topic = ("number", "my_number") #"alias"

 number_publisher_node = Node(
     package="my_cpp_pkg",
     executable="number_publisher",
     name="my_number_publisher", #Remap node name
```

```
        remappings=[                #Remap topics & services
            remap_number_topic
        ],
        parameters=[                #Configurar parámetros
            {"number_to_publish": 4},
            {"publsh_frequency": 1.0}
        ]
    )
```

Al escribir estos archivos con Python, es posible incluir condiciones, bucles y lógica personalizada para iniciar y controlar los nodos. Un archivo de lanzamiento se convierte así en la herramienta maestra para personalizar completamente una aplicación ROS2.

ROS2 Bags

Las '**bags**' son herramientas fundamentales que permiten la captura, almacenamiento y reproducción de datos. Tienen la capacidad de grabar datos de temas durante la ejecución del robot y reproducirlos más tarde de manera idéntica, lo que facilita un análisis detallado y el desarrollo continuo del robot.

Se recomienda crear una carpeta 'bags' y almacenar los archivos grabados en ella para tenerlo más organizado.

Para empezar, necesitamos que algún topic esté en funcionamiento para poder grabarlo. Si existe algún topic funcionando, los comandos para las bags son los siguientes:

```
ros2 bag record /NombreTopic
```

De la manera anterior el nombre del archivo creado lo hará automático, para darle un nombre específico hay que añadir `-o nombreArchivo`

Es posible grabar más de un topic en el mismo bag:

```
ros2 bag record /NombreTopic /NombreTopic2 -o nombreArchivo
```

O grabar todos los topics en funcionamiento:

```
ros2 bag record -a -o nombreArchivo
```

Una vez grabado podemos ver la información del archivo:

```
ros2 bag info nombreArchivo
```

O reproducir los datos grabados

```
ros2 bag play nombreArchivo
```