ROS2 Foxy

Lo primero que se debe hacer es ejecutar el setup.bash, de lo contrario, no se podrá ejecutar nada del ROS2, para eso:

- `source /opt/ros/foxy/setup.bash`

Nodos:

Cada nodo en ROS será responsable de un único propósito (por ejemplo, un nodo para controlar los motores de las ruedas, otro para controlar la información de un Lidar, etc). Cada nodo puede enviar y recibir datos de otros nodos por medio de “topics”, servicios, acciones o parámetros.
Un sistema robótico completo está compuesto de muchos nodos trabajando al tiempo.

El comando **ros2 run** ejecuta un ejecutable de un paquete

- ros2 run <package_name> <executable_name>

Por ejemplo, para correr el nodo de turtlesim:

- ros2 run turtlesim turtlesim_node

ros2 node list: Al ejecutar este comando, se muestran todos los nodos activos; esto es especialmente útil cuando se quiere interactuar con un nodo, o cuando se están ejecutando varios nodos y se les quiere hacer un seguimiento.
Si por ejemplo, se ejecuta el comando *ros2 run turtlesim turtlesim_node* y luego en otro command window, se *ejecuta ros2 node list* se imprimirá en pantalla: /turtlesim.

Remapping: Permite reasignar valores en las propiedades de un nodo, tales como: nombre del nodo, nobre del topic, servicios, nombre, etc.

- ros2 run turtlesim turtlesim_node --ros-args --remap __node:=my_turtle

Al ejecutar el comando anterior, se esta haciendo un remapping para cambiar el nombre del nodo de turtlesim a my_turtle.

Si se tienen dos o más nodos de tortuga ejecutándose, para elegir cual de las dos controlar, se ejecuta:

- ros2 run turtlesim turtle_teleop_key --ros-args --remap turtle1/cmd_vel:=turtle2/cmd_vel

turtle2/cmd_vel, significa que ahora el nodo de cmd_vel será ejecutado sobre turtle2

Luego de conocer el nombre de los nodos con *ros2 node list*, se puede ejecutar

- *ros2 node info <node_name>*

para conocer más información acerca de ese nodo en cuestión. Por ejemplo:

- ros2 node info /my_turtle

Este comando, devuelve una lista de suscriptores, publicadores, servicios y acciones que interactúan con el nodo especificado.

```
/my_turtle
  Subscribers:
    /parameter_events: rcl_interfaces/msg/ParameterEvent
```

```
    /turtle1/cmd_vel: geometry_msgs/msg/Twist
  Publishers:
    /parameter_events: rcl_interfaces/msg/ParameterEvent
    /rosout: rcl_interfaces/msg/Log
    /turtle1/color_sensor: turtlesim/msg/Color
    /turtle1/pose: turtlesim/msg/Pose
  Services:
    /clear: std_srvs/srv/Empty
    /kill: turtlesim/srv/Kill
    /reset: std_srvs/srv/Empty
    /spawn: turtlesim/srv/Spawn
    /turtle1/set_pen: turtlesim/srv/SetPen
    /turtle1/teleport_absolute: turtlesim/srv/TeleportAbsolute
    /turtle1/teleport_relative: turtlesim/srv/TeleportRelative
    /my_turtle/describe_parameters: rcl_interfaces/srv/DescribeParameters
    /my_turtle/get_parameter_types: rcl_interfaces/srv/GetParameterTypes
    /my_turtle/get_parameters: rcl_interfaces/srv/GetParameters
    /my_turtle/list_parameters: rcl_interfaces/srv/ListParameters
    /my_turtle/set_parameters: rcl_interfaces/srv/SetParameters
    /my_turtle/set_parameters_atomically: rcl_interfaces/srv/SetParametersAtomically
  Action Servers:
    /turtle1/rotate_absolute: turtlesim/action/RotateAbsolute
  Action Clients:
```

Topics:

Los topics son un elemento vital para la comunicación por ROS, ellos actúan como buses para que los nodos intercambien mensajes. Un nodo puede publicar información a cualquier número de topics y simultaneamente estar suscrito a otros nodos. Los topics no tienen que trabajar en una comunicación uno a uno, puede ser de uno a muchos, de muchos a uno o de muchos a muchos.

rqt_graph: Este comando se usa para visualizar el intercambio de nodos y topics y las conecciones entre ellos.

ros2 topic echo: Para mirar la información que está siendo publicada a un topic:

- ros2 topic echo <topic_name>

Un ejemplo es:

- ros2 topic echo /turtle1/cmd_vel

El comando anterior, muestra:

```
linear:
  x: 2.0
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
```

z: 0.0
—

ros2 topic info: Otra forma de observar los topic, es con el comando:

- *ros2 topic info /turtle1/cmd_vel*

Lo anterior muestra:

Type: geometry_msgs/msg/Twist
Publisher count: 1
Subscription count: 2

Los nodos envían datos a través de los topics usando mensajes. Los publicadores y los suscriptores deben enviar y recibir el mismo tipo de mensajes para comunicarse.

Si se ejecuta:

- ros2 topic list -t

Se retornará un mensaje mostrando la lista de los topic activos y en corchetes el tipo de mensaje. En el caso del topic cmd_vel (ejemplo del turtlesim), se mostrará entre corchetes [geometry_msgs/msg/Twist], lo que significa que hay un paquete llamado geometry_msgs y existe un mensaje (msg) llamado Twist.

Ejecutando el comando

- *ros2 interface show geometry_msgs/msg/Twist*

  se muestra especificamente la estructura de datos que se espera en el mensaje, la respuesta a este comando sería:

  This expresses velocity in free space broken into its linear and angular parts.

    Vector3 linear
    Vector3 angular

Esto significa que el nodo /turtlesim está esperando un mensaje con dos vectores, uno lineal y otro angular de 3 elementos cada uno.

ros2 publicación en topic

Ahora que se conoce la estructura de los mensajes, se pueden publicar datos directamente en un topic, usando:

- ros2 topic pub <topic_name> <msg_type> '<args>'

<args> corresponde a los datos que se van a enviar al topic

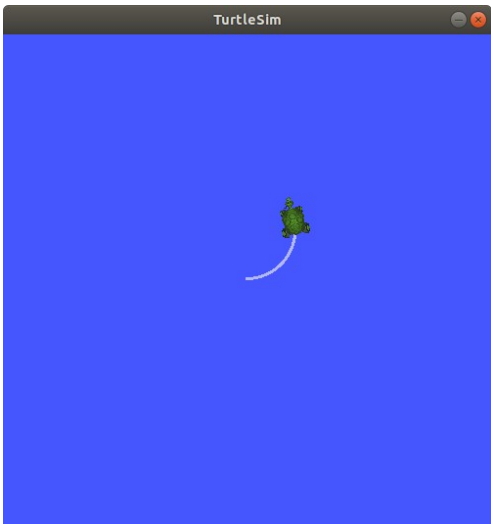
Un ejemplo sería:

- ros2 topic pub --once /turtle1/cmd_vel geometry_msgs/msg/Twist "{linear: {x: 2.0, y: 0.0, z: 0.0}, angular: {x: 0.0, y: 0.0, z: 1.8}}"

La sentencia --once es opcional y significa que el mensaje se publicará una unica vez. En respuesta, se imprimirá el siguiente mensaje en consola:

publisher: beginning loop
publishing #1: geometry_msgs.msg.Twist(linear=geometry_msgs.msg.Vector3(x=2.0, y=0.0, z=0.0), angular=geometry_msgs.msg.Vector3(x=0.0, y=0.0, z=1.8))

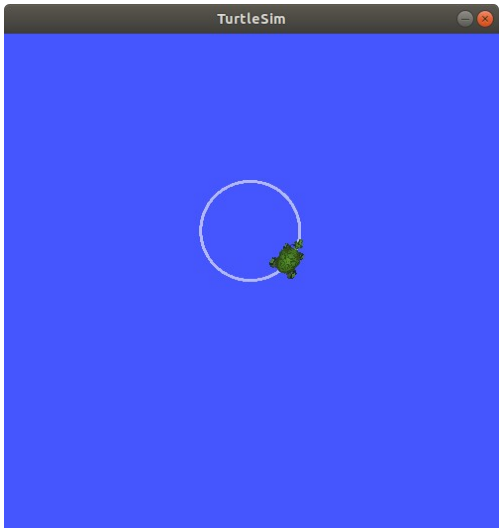
Luego de esto, la tortuga comenzará a moverse según lo indicado.



Otro mensaje similar es:

- ros2 topic pub --rate 1 /turtle1/cmd_vel geometry_msgs/msg/Twist "{linear: {x: 2.0, y: 0.0, z: 0.0}, angular: {x: 0.0, y: 0.0, z: 1.8}}"

La única diferencia con el anterior es *--rate 1*, esta sentencia lo que quiere decir es que se publicará el mensaje con una frecuencia de 1Hz, lo que en este caso se reflejará en una trayectoria circular descrita por la tortuga.



NOTA: Algo muy importante a tener en cuenta con el pub, es que se deben respetar espacios; siempre dejar espacio luego de los dos puntos ":" y tras una coma "," que significa cambio de posición en los vectores (si no se respeta el espacio, no funciona).

ros2 topic hz: Permite visualizar la tasa en la cual los datos se están publicando sobre un topic.

- Ros2 topic hz turtle/pose

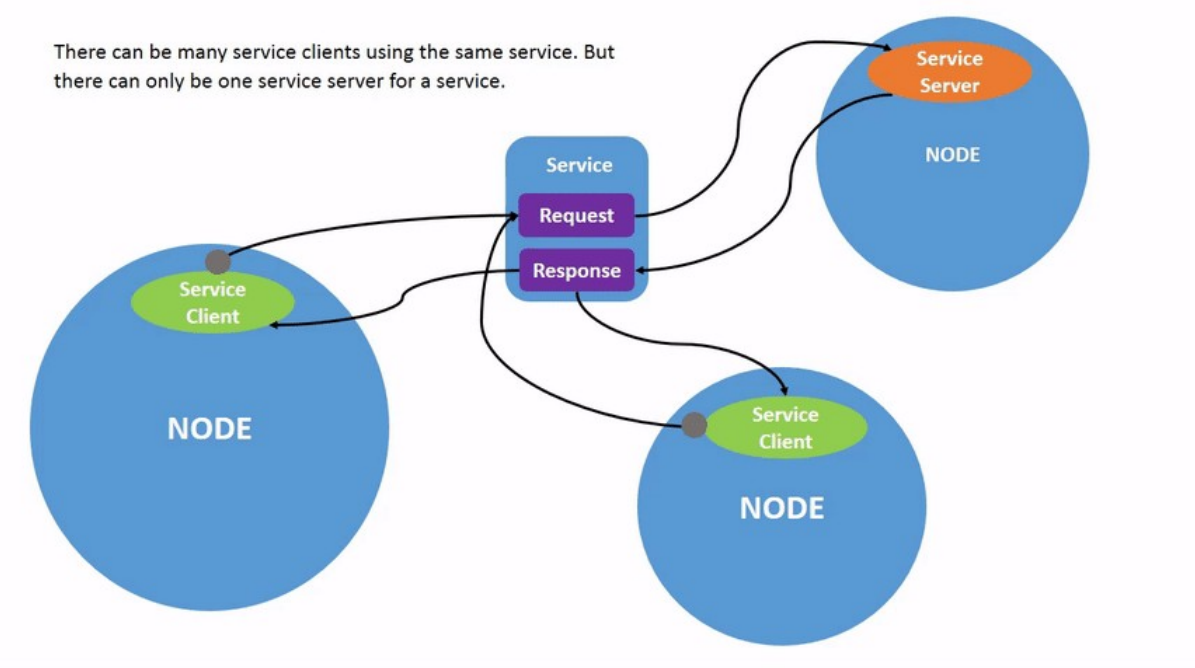
Esto retornará el siguiente mensaje:

```
average rate: 59.354
  min: 0.005s max: 0.027s std dev: 0.00284s window: 58
```

Services:

Los servicios son otra forma de comunicación en ROS, se basan en un sistema "llamada-respuesta", a diferencia de los publicadores y suscriptores en los topics.
La mayor diferencia con los topics es que los servicios solo proporcionan información o "datos" cuando se les solicita, mientras que los topics comparten la infromación con los nodos suscritos sin necesidad de solicitarla.

Para emplear servicios, existen nodos clientes y nodos servidores; pueden existir varios clientes usando el mismo servicio, pero solo puede existir un server por servicio, es decir, solo puede haber un nodo tipo server.



ros2 service list: ejecutar el comando *ros2 service list* en una terminal luego de habilitar los nodos *turtlesim_node* y *turtle_teleop_key* se mostrará en la terminal lo siguiente:

/clear
/kill
/reset
/spawn
/teleop_turtle/describe_parameters
/teleop_turtle/get_parameter_types
/teleop_turtle/get_parameters
/teleop_turtle/list_parameters
/teleop_turtle/set_parameters
/teleop_turtle/set_parameters_atomically
/turtle1/set_pen
/turtle1/teleport_absolute
/turtle1/teleport_relative
/turtlesim/describe_parameters
/turtlesim/get_parameter_types
/turtlesim/get_parameters
/turtlesim/list_parameters
/turtlesim/set_parameters
/turtlesim/set_parameters_atomically

se pude notar que tanto el nodo del *tuttlesim_node* como *turtle_teleop_key* tienen los mismos “parámetros”, posteriormente se hablará mejor acerca de esto.

De momento, se hará énfasis en lls servicios: /*clear, /kill, /reset, /spawn, /turtle1/set_pen, turtle1/teleport_absolute, /turtle1/teleport_relative*. Se puede emplear la interfaz de rqt para interactuar con estos servicios.

ros2 service types: Los servicios tienen tipos, los cuáles describen como está estructurado el servicio. Los servicios tienen una estructura similar a los topics, con la diferencia de que los servicios tienen dos partes, un mensaje de *request* y otro de *response*.

Para descubrir el tipo de servicio, se escribe el comando:

- ros2 service type <service_name>

Un ejemplo de este comando con el servicio /clear sería:

- ros2 service type /clear

Lo anterior retorna un mensaje en consola:

std_srvs/srv/Empty

El tipo “Empty” significa que el servicio no envía datos cuando se le hace una solicitud “request”, y tampoco envía ningún dato cuando se recibe una respuesta “response”.

ros2 service list -t: Al ejecutar el comando *ros2 service list -t* se imprime en pantalla:

```
/clear [std_srvs/srv/Empty]
/kill [turtlesim/srv/Kill]
/reset [std_srvs/srv/Empty]
/spawn [turtlesim/srv/Spawn]
...
/turtle1/set_pen [turtlesim/srv/SetPen]
/turtle1/teleport_absolute [turtlesim/srv/TeleportAbsolute]
/turtle1/teleport_relative [turtlesim/srv/TeleportRelative]
…
```

Se puede notar que funciona similar al comando *ros2 topic list -t*, entre corchetes se muestra información acerca del servicio.

ros2 service find: Para visualizar todos los servicios de un tipo específico, se ejecuta

- *ros2 service find <type_name>*

Un ejemplo de lo anterior sería:

- ros2 service find std_srvs/srv/Empty

Lo cual devuelve en pantalla:

/clear
/reset

ros2 interface show: Se pueden llamar servicios desde una línea de comando, sin embargo, primero se debe conocer la estructura de los argumentos de entrada. *ros2 interface show <type_name>.srv*

Si se ejecuta:

- ros2 interface show std_srvs/srv/Empty.srv

Lo cual retornará:

Las líneas --- separan la estructura de request (segmento superior) de la estructura de response (segmento inferior). Pero como se mencionó anteriormente, el tipo “Empty” no envía ni recibe mensajes, es por eso, que la estructura aparece en blanco.

Un servicio que sí envía y recibe mensajes es /spawn, un ejemplo con este servicio sería:

- ros2 interface show turtlesim/srv/Spawn.srv

Lo cual retorna:

```
float32 x
float32 y
float32 theta
string name # Optional.  A unique name will be created and returned if this is empty
---
string name
```

ros2 service call: Ahora que se sabe el tipo de servicio, como buscar el tipo de servicio y como conocer la estructura de los argumentos del servicio, se puede llamar.

- ros2 service call <service_name> <service_type> <arguments>

Cabe aclarar que la parte de los argumentos es opcional, no es necesario definirlos para llamar al servicio.

Un ejemplo del service call:

- ros2 service call /spawn turtlesim/srv/Spawn "{x: 2, y: 2, theta: 0.2, name: 'Mbappe'}"

Se hará un *spawn* de una tortuga llamada “Mbappe” en la posición indicada.

Parámetros:

Un parámetro es la configuración del valor en un nodo, se puede pensar en los parámetros como “configuraciones de nodo”. Un nodo puede guardar parámetros en forma de : integers, floats, booleans, strings y lists.

ros2 param list: Al ejecutar el comando *ros2 param list* luego de tener abiertos los nodos de *turtlesim_node* y *turtle_teleop_key*, se mostrará en pantalla:

```
/teleop_turtle:
  scale_angular
  scale_linear
  use_sim_time
/turtlesim:
  background_b
  background_g
  background_r
  use_sim_time
```

Se puede notar que */turtlesim* permite modificar el color del fondo especificando el patrón RGB.

ros2 param get: Para mostrar en pantalla el tipo y el valor actual de un parámetro, se ejecuta:

- ros2 param get <node_name> <parameter_name>

Por ejemplo, al ejecutar *ros2 param get /turtlesim background_g*, se imprimirá en pantalla:

```
Integer value is: 86
```

Esto quiere decir, que el valor de verde en el fondo es de 86.

ros2 param set: Para cambiar el valor de algún parámetro, se ejecuta:

- ros2 param set <node_name> <parameter_name> <value>

Un ejemplo, sería ejecutar: *ros2 param set /turtlesim background_r 150*, lo cual devuelve el siguiente mensaje en la terminal:

```
ros2 param set /turtlesim background_r 150
```

Y el color de fondo efectivamente habrá cambiado.

ros2 param dump: Se pueden “volcar” todos los valores de los parámetros dentro de un archivo y guardarlos para usarlos después en otros comandos. Para eso:

- ros2 param dump <node_name>

Un ejemplo es ejecutar: *ros2 param dump /turtlesim* lo cual guardará un archivo en el diredctorio activo del workspace. Si se abre el archivo, se podrá visualizar el nombre del nodo, y una lista de los

parámetros con los valores registrados al momento de guardar la información. Esta función es muy útil si se desean cargar los valores del nodo a los valores de un punto anterior.

ros2 param load: Se pueden cargar los parámetros de un archivo (como el generado en el dump), usando el comando:

- ros2 param load <node_name> <parameter_file>

Un ejemplo de esto, sería ejecutar: *ros2 param load /turtlesim ./turtlesim.yaml*
Lo anterior retornará un mensaje como:

Set parameter background_b successful
Set parameter background_g successful
Set parameter background_r successful
Set parameter use_sim_time successful

Cargar un archivo de parámetros en la inicialización de un nodo: Esto permite definir los parámetros de un nodo en la misma línea de código en el cual se está inicializando desde un archivo de parámetro, para esto se ejecuta:

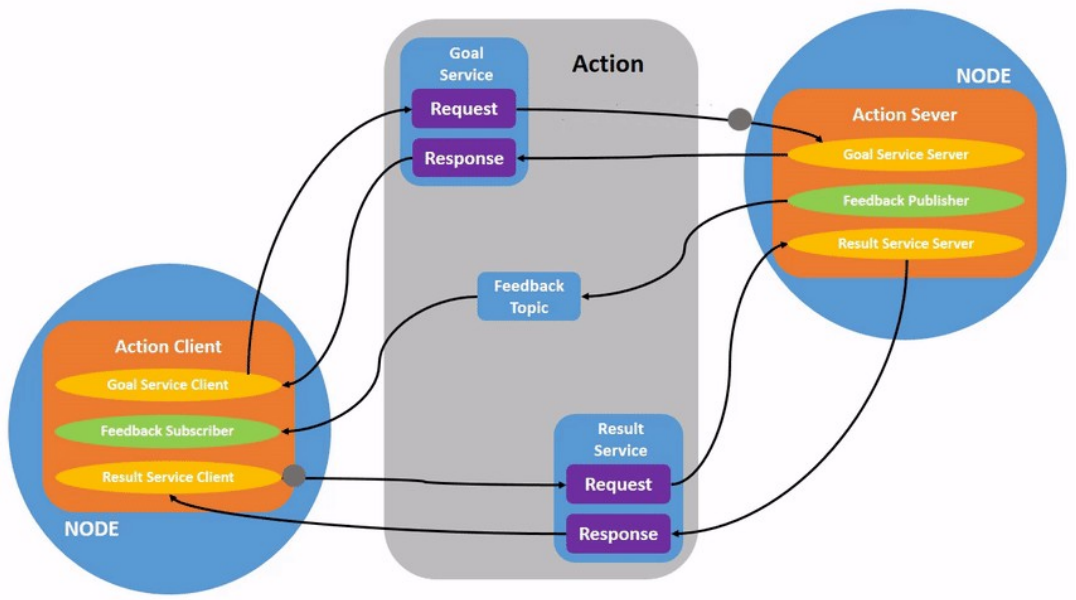
- ros2 run <package_name> <executable_name> --ros-args --params-file <file_name>

Un ejemplo de esto, sería ejecutar: *ros2 run turtlesim turtlesim_node --ros-args --param-file ./turtlesim.yaml*

Con esto, se ejecutará el nodo del turtlesim como siempre, pero con el color de fondo modificado (ya que previamente se “modificó” el parámetro y se guardó en el archivo turtlesim.yaml.

Actions:

Las acciones funcionan similar a los servicios, con la diferencia de que se pueden cancelar mientras se ejecutan; las acciones usan un servicio de cliente-servidor, similar al modelo publisher-subscriber, un nodo “action client” envía una meta a un nodo “action server” que reconoce la meta y retorna una retroalilmentación y un resultado.



Use actions: Cuando se ejecuta el nodo *teleop_turtle*, se verá la siguiente información en pantalla:

Use arrow keys to move the turtle.
Use G|B|V|C|D|E|R|T keys to rotate to absolute orientations. 'F' to cancel a rotation.

Nos centraremos en la segunda línea que corresponde a una acción. Las letras que se muestran en esta línea corresponden a una rotación absoluta de la tortuga en el plano, en el nodo del *turtleisim_node* se verá el siguiente mensaje luego de completar la rotación satisfactoriamente:

[INFO] [turtlesim]: Rotation goal completed successfully

La letra ‘F’ se usa para cortar la rotación, en donde se demuestra que las acciones pueden ser canceladas, y al presionar la tecla se imprimirá el siguiente mensaje en pantalla:

[INFO] [turtlesim]: Rotation goal canceled

Si se ejecuta el comando

- ros2 node info /turtlesim (ya previamente se explicó esto en la parte de los nodos)

Se mostrará en pantalla:

```
/turtlesim
  Subscribers:
    /parameter_events: rcl_interfaces/msg/ParameterEvent
    /turtle1/cmd_vel: geometry_msgs/msg/Twist
  Publishers:
    /parameter_events: rcl_interfaces/msg/ParameterEvent
    /rosout: rcl_interfaces/msg/Log
    /turtle1/color_sensor: turtlesim/msg/Color
```

```
  /turtle1/pose: turtlesim/msg/Pose
 Services:
  /clear: std_srvs/srv/Empty
  /kill: turtlesim/srv/Kill
  /reset: std_srvs/srv/Empty
  /spawn: turtlesim/srv/Spawn
  /turtle1/set_pen: turtlesim/srv/SetPen
  /turtle1/teleport_absolute: turtlesim/srv/TeleportAbsolute
  /turtle1/teleport_relative: turtlesim/srv/TeleportRelative
  /turtlesim/describe_parameters: rcl_interfaces/srv/DescribeParameters
  /turtlesim/get_parameter_types: rcl_interfaces/srv/GetParameterTypes
  /turtlesim/get_parameters: rcl_interfaces/srv/GetParameters
  /turtlesim/list_parameters: rcl_interfaces/srv/ListParameters
  /turtlesim/set_parameters: rcl_interfaces/srv/SetParameters
  /turtlesim/set_parameters_atomically: rcl_interfaces/srv/SetParametersAtomically
 Action Servers:
  /turtle1/rotate_absolute: turtlesim/action/RotateAbsolute
 Action Clients:
```

Se puede notar que la acción */turtle1/rotate_absolute* del nodo */turtlesim* está debajo de “Action Servers”, esto significa que el nodo */turtlesim* responde y provee realimentación para la acción */turtle1/rotate_absolute.*
Por otro lado, si se ejecuta el comando *info* para el nodo */teleop_turtle* la acción */turtle1/rotate_absolute* está en la parte de “Action Clients”, lo que significa que es este nodo el que envía la meta para la acción, tal que:

```
/teleop_turtle
 Subscribers:
  /parameter_events: rcl_interfaces/msg/ParameterEvent
 Publishers:
  /parameter_events: rcl_interfaces/msg/ParameterEvent
  /rosout: rcl_interfaces/msg/Log
  /turtle1/cmd_vel: geometry_msgs/msg/Twist
 Services:
  /teleop_turtle/describe_parameters: rcl_interfaces/srv/DescribeParameters
  /teleop_turtle/get_parameter_types: rcl_interfaces/srv/GetParameterTypes
  /teleop_turtle/get_parameters: rcl_interfaces/srv/GetParameters
  /teleop_turtle/list_parameters: rcl_interfaces/srv/ListParameters
  /teleop_turtle/set_parameters: rcl_interfaces/srv/SetParameters
  /teleop_turtle/set_parameters_atomically: rcl_interfaces/srv/SetParametersAtomically
 Action Servers:

 Action Clients:
  /turtle1/rotate_absolute: turtlesim/action/RotateAbsolute
```

ros2 action list: Para identificar todas las acciones corriendo en ROS, se ejecuta el comando:

- ros2 action list

Lo cual devolverá en pantalla (si se tienen activos los nodos de teleop y turtlesim):

/turtle1/rotate_absolute

ros2 action list -t: Las acciones tienen tipos, como los topics, al ejecutar el comando:

- ros2 action list -t

Se retornará en pantalla:

/turtle1/rotate_absolute [turtlesim/action/RotateAbsolute]

Entre los corchetes se encuentra el tipo de acción.

ros2 action info: Para obtener información de las acciones, se ejecuta el comando:

- ros2 action info <action_name>

Un ejemplo sería ejecutar: *ros2 action info /turtle/rotate_absolute*

Lo cual retornará en pantalla:

```
Action: /turtle1/rotate_absolute
Action clients: 1
   /teleop_turtle
Action servers: 1
   /turtlesim
```

Esto nos indica que para la acción */turtle1/rotate_absolute* en “Action clients” tiene al nodo */teleop_turtle,* y en “Action servers” está el nodo */turtlesim*

ros2 interface show: Similar al interface show para topics, o services, existe el comando para las acciones. Por ejemplo si se ejecuta:

- *ros2 interface show turtlesim/action/RotateAbsolute*.

Se muestra en pantalla:

```
The desired heading in radians
float32 theta
---
The angular displacement in radians to the starting position
float32 delta
---
The remaining rotation in radians
float32 remaining
```

La primera sección arriba de --- es la solicitud de meta, la segunda sección es la estructura del resultado, y la última sección es la estructura de la realimentación.

ros2 action send_goal: Ahora, luego de conocer la estructura de los mensajes de la acción, se le puede enviar una meta.

- ros2 action send_goal <action_name> <action_type> <values>

LOS VALORES (<values>) DEBEN ESTAR EN FORMATO .YAML

Un ejemplo de lo anterior es:

- *ros2 action send_goal /turtle1/rotate_absolute turtlesim/action/RotateAbsolute "{theta: 1.57}"*

Se mostrará en pantalla:

Waiting for an action server to become available...
Sending goal:
  theta: 1.57

Goal accepted with ID: f8db8f44410849eaa93d3feb747dd444

Result:
  delta: -1.568000316619873

Goal finished with status: SUCCEEDED

Todas las acciones tienen una ID única. Se puede mirar el resultado, indicado como “delta” en este caso. Para ver la realimentación, se debe añadir --feedback al último comando mostrado, tal que:

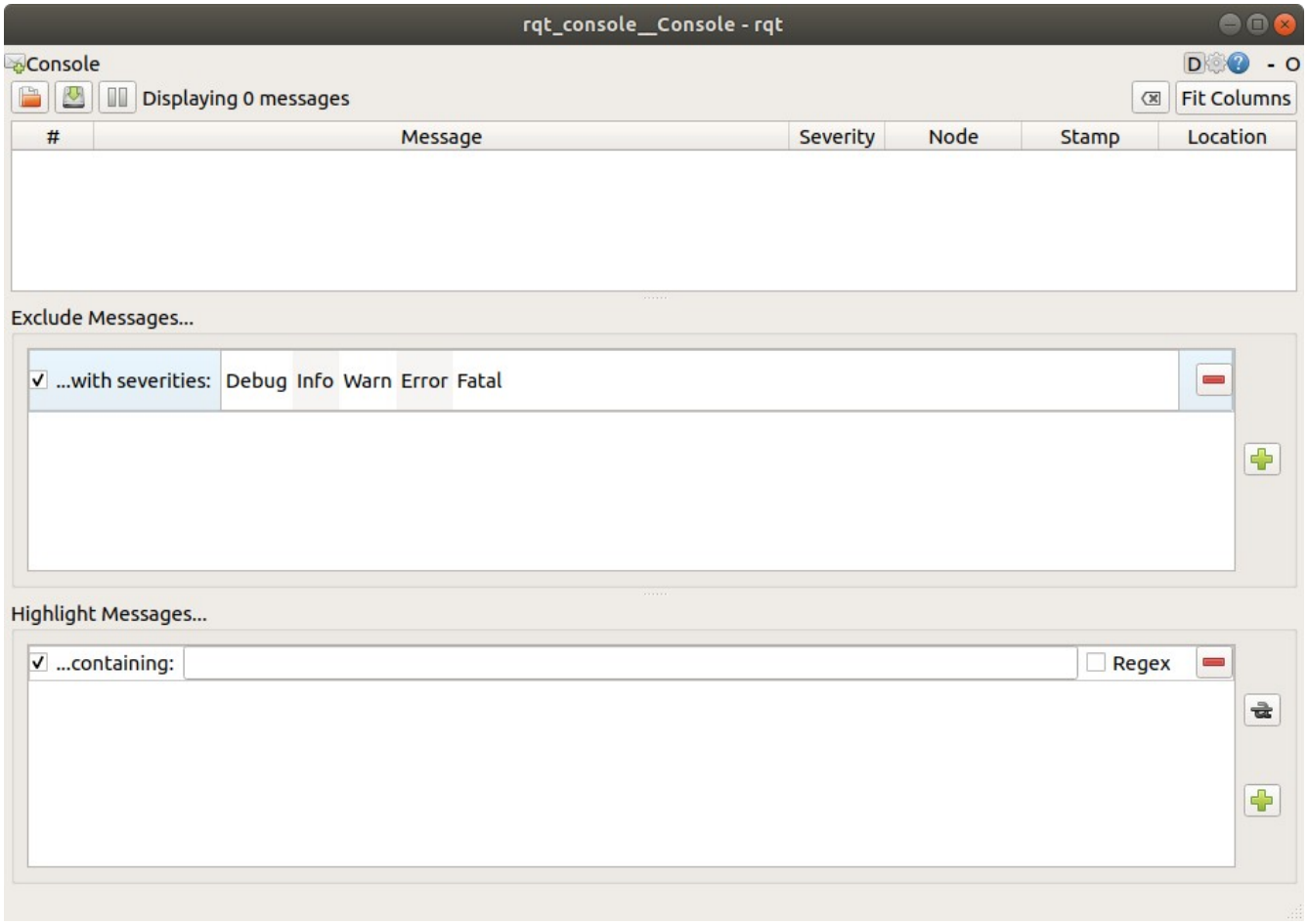
- ros2 action send_goal /turtle1/rotate_absolute turtlesim/action/RotateAbsolute "{theta: 1.57}" –feedback

El feedback mostrará un mensaje por cada rotación que realice hasta llegar a la meta definida (1.57rad en este caso).

Using rqt_console to view logs

Para abrir la ventana del rqt_console, se ejecuta:

- ros2 run rqt_console rqt_console



La primera sección de la consola es donde se pueden ver los mensajes de “log”. En la sección del medio se tiene la opción de filtrar los mensajes. La última sección resalta los mensajes con el “String” que se defina como filtro.

Logger levels: Los logger levels están ordenados por severidad:

1. Fatal: Este tipo de mensajes indican que se va a detener para intentar protegerse de dasños.
2. Error: Indica problemas significativos pero que no necesariamente van a dañar el sistema, pero sí impiden que funcione correctamente.
3. Warn: Indican una actividad no esperada o un resultado no ideal que pueden representar un problema más profundo.
4. Info: Indica actualizaciones de estado o de evento que sirven como una verificación visual de que el sistema está ejecutándose como se espera.
5. Debug: Mensajes que indican a detalle el paso a paso del proceso para la ejecución del sistema.

El sistema solo mostrará los mensajes configurados en “default” y de orden superior al indicado; ROS2 por defecto trae a “Info” como el nivel “defaul”, entonces solo se verán en pantalla los mensajes de tipo Info, Warn, Error y Fatal. Para cambiar el tipo de mensaje “default”, se ejecuta:

- ros2 run turtlesim turtlesim_node --ros-args --log-level WARN

La línea anterior indica que en el modo default se establecerán los mensajes WARN, lo que da a entender que los mensajes Info y Debug no serán mostrados.

Launching nodes

Cuando se crean sistemas complejos, no es eficiente tener una terminal abierta por cada nodo en ejecución; las “Launch Files” permiten configurar archivos ejecutables que contienen nodos de ROS 2, basta con ejecutar un solo archivo del tipo *ros2 launch* para que empiece a correr todo el sistema entero.

Si se ejecuta el comando:

- ros2 launch turtlesim multisim.launch.py

El archivo .launch que se ejecuta es el siguiente script:

```
# turtlesim/launch/multisim.launch.py

from launch import LaunchDescription
import launch_ros.actions

def generate_launch_description():
   return LaunchDescription([
      launch_ros.actions.Node(
         namespace= "turtlesim1", package='turtlesim', executable='turtlesim_node', output='screen'),
      launch_ros.actions.Node(
         namespace= "turtlesim2", package='turtlesim', executable='turtlesim_node', output='screen'),
   ])
```

El .launch anterior abrirá dos ventanas de turtlesim.

Ahora con las dos ventanas abiertas, se pueden mover publicando mensajes en el topic del cmd_vel. Para la tortuga1:

- ros2 topic pub /turtlesim1/turtle1/cmd_vel geometry_msgs/msg/Twist "{linear: {x: 2.0, y: 0.0, z: 0.0}, angular: {x: 0.0, y: 0.0, z: 1.8}}"

Para la tortuga 2:

- ros2 topic pub /turtlesim2/turtle1/cmd_vel geometry_msgs/msg/Twist "{linear: {x: 2.0, y: 0.0, z: 0.0}, angular: {x: 0.0, y: 0.0, z: -1.8}}".

De momento solo es un abre-bocas de cómo funcionan los archivos .launch, más adelante se aprenderá a escribir los propios.

Recording and playing back data

ros2 bag es una línea de comando para grabar los dastos publicados en los topics en el ordenador. Acumula los datos de cualquier número de topics y los guarda en una base de datos, posteriormente se pueden reproducir los resultados del test y de los experimentos. Grabar los topics también es una gran forma de compartir tu trabajo.

Antes de grabar la información publicada en un topic, se debe seleccionar uno, para eso, se puede emplear el comando ya visto *ros2 topic list*

ros2 bag record: Para grabar los dsatos publicados en un topic, se emplea:

- ros2 bag record <topic_name>

Un ejemplo de lo anterior, sería:
ros2 bag record /turtle1/cmd_vel
Lo cual empezaría a grabar los datos del topic cmd_vel

Se imprimirán los siguiente mensajes en la pantalla:

[INFO] [rosbag2_storage]: Opened database 'rosbag2_2019_10_11-05_18_45'.
[INFO] [rosbag2_transport]: Listening for topics...
[INFO] [rosbag2_transport]: Subscribed to topic '/turtle1/cmd_vel'
[INFO] [rosbag2_transport]: All requested topics are subscribed. Stopping discovery…

Ahora, si se empieza a mover la tortuga empleando el nodo cmd_vel, los datos se empezarán a grabar.

Record multiple topics:

Para grabar varios topics al tiempo, se ejecuta el comando anterior, y se especifican los diferentes topics con un espaciado entre ellos.

Un ejemplo:

ros2 bag record -o subset /turtle1/cmd_vel /turtle1/pose

En el comando anterior, *-o subset*, significa que se va a asignar un nombre “-o” y el nombre será “subset”.

Otra opción es añadir *-a* lo que significa que se grabarán todos los topics activos en el sistema.

ros2 bag info: Para visualizar los detalles grabados, se ejecuta:

- ros2 bag info <bag_file_name>

Un ejemplo de esto:

ros2 bag info subset

Lo cual mostrará en pantalla:

```
Files:             subset.db3
Bag size:          228.5 KiB
Storage id:        sqlite3
Duration:          48.47s
Start:             Oct 11 2019 06:09:09.12 (1570799349.12)
End                Oct 11 2019 06:09:57.60 (1570799397.60)
Messages:          3013
Topic information: Topic: /turtle1/cmd_vel | Type: geometry_msgs/msg/Twist | Count: 9 | Serialization Format: cdr
                   Topic: /turtle1/pose | Type: turtlesim/msg/Pose | Count: 3004 | Serialization Format: cdr
```

ros2 bag play: Antes de reproducir el archivo bag, se deben dejar de ejecutar los nodos que publican sobre los topics, luego se ejecuta:

- ros2 bag play subset

Como se grabaron los mensajes del topic cmd_vel, se debe tener visibilidad de la ventana con la tortuga para ver como reproduce los movimientos.

Creating a workspace (ws)

Para estabecer un espacio de trabajo, se debe crear primero una carpeta y dentro de ella, otra carpeta llamada src, el nombre de la carpeta “raíz”, por lo general se denota por nombreCarpeta_ws

Se chequean las dependencias:

- rosdep install -i --from-path src --rosdistro foxy -y

Ahora se puede hacer el build en la raíz del workspace:

- colcon build

Luego de ejecutar el build, habrán 4 carpetas dentro del workspace: build install log src

Ahora se debe hacer el source, no obstante, no se debe hacer en la misma terminal en donde se creó el workspace, esto puede traer muchos problemas, por eso, se abre una nueva terminal y se digita:

- source /opt/ros/foxy/setup.bash

Ahora en esa nueva terminal nos dirigimos a la carpeta raíz del workspace.

Luego de estar dentro de esta carpeta, se va a crear un source local, para ello:

- . install/local_setup.bash

Creating a package

Un paquete es una especie de contenedor para el código de ROS 2. Si se quiere instalar el código o compartirlo, es necesario que esté organizado en un paquete, con un paquete se puede "liberar" el trabajo de ROS 2 y permite que cualquiera pueda hacer un build y usarlo fácilmente. Los paquetes se deben crear dentro de un ws activo.

El comando para crear un nuevo paquete en python es:

- ros2 pkg create --build-type ament_python <package_name>

Para crear el paquete en CMake:

- ros2 pkg create --build-type ament_cmake <package_name>

Tanto en Python como en CMake se puede definir el nombre del nodo agregando la sentencia a la línea de código:

- ros2 pkg create --build-type ament_python --node-name my_node my_package
- ros2 pkg create --build-type ament_cmake --node-name my_node my_package

Luego de hacer eso, deberá tener una nueva carpeta dentro de src con el nombre del paquete (en este caso *my_package*). Además, se imprimirá en la terminal:

```
going to create a new package
package name: my_package
destination directory: /home/user/ros2_ws/src  //La ubicación depende de donde se haya creado el ws
package format: 3
version: 0.0.0
description: TODO: Package description
maintainer: ['<name> <email>']
licenses: ['TODO: License declaration']
build type: ament_cmake
dependencies: []
node_name: my_node
creating folder ./my_package
creating ./my_package/package.xml
```

```
creating source and include folder
creating folder ./my_package/src
creating folder ./my_package/include/my_package
creating ./my_package/CMakeLists.txt
creating ./my_package/src/my_node.cpp
```

Build a package: Tener muchos paquetes dentro de un ws es muy útil ya que se puede hacer un *build* de todos a la misma vez empleando *colcon build en la raíz del ws.*

Source the setup file: Para usar el nuevo paquete y el ejecutable, primero se deberá abrir una nueva terminal y hacer el source general:

- source /opt/ros/foxy/setup.bash

Luego dentro del ws se corre el siguiente comando para el source local:

- . install/setup.bash

Ahora se puede correr el paquete creado:

- ros2 run my_package my_node

Customize package.xml: En el mensaje que se imprime luego de la creación del paquete, se puede notar que los campos *description* & *license* contienen notas *T0D0*, eso se debe a que la descripción del paquete y la licencia no se declaran de forma automática, pero son necesarias si se quiere "liberar" el paquete, tambień se tiene que diligenciar el campo de *maintainer*.

Desde la carpeta del paquete (ubicada dentro de src en el ws) abra el archivo llamado *package.xml* con el editor de texto preferido.

Para paquetes creados con CMake:

```
<?xml version="1.0"?>
<?xml-model
  href="http://download.ros.org/schema/package_format3.xsd"
  schematypens="http://www.w3.org/2001/XMLSchema"?>
<package format="3">
 <name>my_package</name>
 <version>0.0.0</version>
 <description>TODO: Package description</description>
 <maintainer email="user@todo.todo">user</maintainer>
 <license>TODO: License declaration</license>

 <buildtool_depend>ament_cmake</buildtool_depend>
```

```
 <test_depend>ament_lint_auto</test_depend>
 <test_depend>ament_lint_common</test_depend>

 <export>
   <build_type>ament_cmake</build_type>
 </export>
</package>
```

Para paquetes creados con Python:

```
<?xml version="1.0"?>
<?xml-model
   href="http://download.ros.org/schema/package_format3.xsd"
   schematypens="http://www.w3.org/2001/XMLSchema"?>
<package format="3">
 <name>my_package</name>
 <version>0.0.0</version>
 <description>TODO: Package description</description>
 <maintainer email="user@todo.todo">user</maintainer>
 <license>TODO: License declaration</license>

 <test_depend>ament_copyright</test_depend>
 <test_depend>ament_flake8</test_depend>
 <test_depend>ament_pep257</test_depend>
 <test_depend>python3-pytest</test_depend>

 <export>
   <build_type>ament_python</build_type>
 </export>
</package>
```

En la parte de *maintainer* debe poner su nombre y su correo, se debe dar una breve descripción del paquete, y en la parte de la licencia debe especificar el tipo de licencia (para eso investigue sobre los tipos de licencia para librerías open source aquí

De momento se puede usar la licencia *Apache License 2.0* ya que se trata de una “práctica”.

Writing a simple publisher and subscriber (C++)

Para el tutorial, se crea un paquete dentro del ws:

- ros2 pkg create --build-type ament_cmake cpp_pubsub

Como se creó un paquete nuevo, se tiene que hacer el proceso del *colcon build*, del source general, y el source local.

Para escribir el nodo publisher, se descargara el ejemplo del talker dentro de la carpeta src del paquete creado.

- wget -O publisher_member_function.cpp https://raw.githubusercontent.com/ros2/examples/foxy/rclcpp/topics/minimal_publisher/member_function.cpp

Se creará un archivo .cpp dentro de la carpeta src del paquete, este archivo se puede abrir con el editor de texto de preferencia.

Examinando el código del nodo:

```
#include <chrono>
#include <functional>
#include <memory>
#include <string>

#include "rclcpp/rclcpp.hpp"
#include "std_msgs/msg/string.hpp"

using namespace std::chrono_literals;
```

En esta primera parte, se incluyen los headers estándar de C++ que se estarán usando, luego se encuentran los headers que permitirán usar los sistemas de ROS2 *rclcpp/rclcpp.hpp y* por último se incluirá la construcción de los tipos de mensaje para publicar datos *std_msgs/msg/string.hpp*

```
class MinimalPublisher : public rclcpp::Node
```

Se crea la clase *Minimal Publisher* que herada de *rclcpp::Node*.

El constructor público llama al nodo *minimal_publisher* e inicializa un contador *count_* en cero. Dentro del constructor, el pulbisher es inicializado con mensajes de tipo *String*, el nombre del topic, llamado *topic*, y el tamaño de cola requerido para limitar los mensajes en caso de una copia de seguridad *(“topic” , 10);*
Luego, se inicializa el timer *timer_* el cual ejecuta la función cada 500ms.

```
public:
  MinimalPublisher()
  : Node("minimal_publisher"), count_(0)
  {
    publisher_ = this->create_publisher<std_msgs::msg::String>("topic", 10);
    timer_ = this->create_wall_timer(
    500ms, std::bind(&MinimalPublisher::timer_callback, this));
  }
```

La función *timer_callback* es donde se publican los datos. La parte de *RCLCPP_INFO* es la que se encarga de que los mensajes se publiquen en consola.

```
private:
  void timer_callback()
  {
    auto message = std_msgs::msg::String();
    message.data = "Hello, world! " + std::to_string(count_++);
    RCLCPP_INFO(this->get_logger(), "Publishing: '%s'", message.data.c_str());
    publisher_->publish(message);
  }
```

Por último, se declaran los campos del timer, publisher y contador.
La clase *main* en donde se ejecuta el nodo es *MinimalPublisher*. rclcpp::init inicializa ROS 2 y rclcpp::spin empieza a procesar los datos del nodo, incluídos los *callbacks* del timer.

```
int main(int argc, char * argv[])
{
  rclcpp::init(argc, argv);
  rclcpp::spin(std::make_shared<MinimalPublisher>());
  rclcpp::shutdown();
  return 0;
}
```

Add dependencies: Navegar hacia un directorio atrás, en donde se encuentran los archivos *CmakeLists.txt* y *package.xml*, y abrir este último archivo .xml con el editor de texto y llenar los campos de <depend></depend> de la siguiente forma (basado en las librerías de ROS que se llamaron en el nodo):

```
<depend>rclcpp</depend>
<depend>std_msgs</depend>
```

Esto indica cuales son las dependencias de ROS 2 empleadas *rclpp* y *std_msgs*.

CmakeLists.txt: Ahora abra el archivo CMakeLists.txt y debajo de la dependencia *find_package(ament_cmake REQUIRED), añadir también las dependencias:*

```
find_package(rclcpp REQUIRED)
find_package(std_msgs REQUIRED)
```

Luego añade el ejecutable llamado *talker* así se podrá correr el nodo usando el comando *ros2 run*.

```
add_executable(talker src/publisher_member_function.cpp)
ament_target_dependencies(talker rclcpp std_msgs)
```

Finalmente, agregue la sección *install(TARGETS…)* así *ros2 run* podrá encontrar el ejecutable

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

Ahora se puede hacer un build en el paquete y se puede hacer el source para los archivos locales y correrlo.
Lo siguiente es crear el nodo subscriber

Write the subscriber node: Se debe dirigir a la carpeta src del paquete y usar:

- wget -O subscriber_member_function.cpp https://raw.githubusercontent.com/ros2/examples/foxy/rclcpp/topics/minimal_subscriber/member_function.cpp

Esto creará un archivo .cpp que puede abrirse con cualquier editor de texto. El nodo del subscriber es muy parecido al nodo del publisher, la diferencia más marcada es que este nodo no tiene un timer, lo cuál tiene lógica, ya que este nodo solo responde cuando se le envían los datos a través del topic.

Cabe recordar que para que dos nodos (publisher y subscriber) se puedan entender, es necesario que tengan el mismo tipo de mensaje, en este caso, ambos tienen la estructura de mensajes: s*td_msgs::msg::String*

Como este archivo (subscriber) tienen las mismas dependencias que el publisher, no es necesario modificar el archivo .xml, sin embargo, es necesario agregar algunas cosas del CMakeLists.txt

CMakeLists.txt: Se debe agregar:

```
add_executable(listener src/subscriber_member_function.cpp)
ament_target_dependencies(listener rclcpp std_msgs)

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

Luego de guardar el archivo, ya estará listo para usarse el sistema pub/sub.

Build and run: Como ya se hizo el build del paquete, las dependencias rclcpp y std_msgs deberían estar instaladas en el ws local, no obstante, es una buena práctica dirigirse a la raíz del ws y ejecutar:

- rosdep install -i --from-path src --rosdistro foxy -y

Ahora desde la raíz del ws, se debe hacer el build nuevamente del paquete para actualizar todos los cambios:

- colcon build --packages-select cpp_pubsub

En el comando anterior, se está especificando que el build se hace sobre el paquete cpp_pubsub.

Por último, se debe hacer el source local:

- . install/setup.bash

Ya se puede ejecutar el paquete:

- ros2 run cpp_pubsub talker

En otra terminal hacer un build y un source y ejecutar:

- ros2 run cpp_pubsub listener

Writing a simple service and client (C++):

Se inicia creando un paquete :

- ros2 pkg create --build-type ament_cmake cpp_srvcli --dependencies rclcpp example_interfaces

La parte de --dependencies especifica las dependencias del paquete, y se ponen automáticamente en el archivo CMakeLists.txt.

Se debe actualizar la información del archivo *package.xml* (email, nombre y licencia).

Dentro de la carpeta src del paquete se debe crear el *service node* con el nombre add_two_ints_server.cpp el cual contiene:

```
#include "rclcpp/rclcpp.hpp"
#include "example_interfaces/srv/add_two_ints.hpp"

#include <memory>

void add(const std::shared_ptr<example_interfaces::srv::AddTwoInts::Request> request,
         std::shared_ptr<example_interfaces::srv::AddTwoInts::Response>      response)
{
  response->sum = request->a + request->b;
  RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "Incoming request\na: %ld" " b: %ld",
              request->a, request->b);
  RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "sending back response: [%ld]", (long int)response-
>sum);
}

int main(int argc, char **argv)
{
  rclcpp::init(argc, argv);

  std::shared_ptr<rclcpp::Node> node = rclcpp::Node::make_shared("add_two_ints_server");

  rclcpp::Service<example_interfaces::srv::AddTwoInts>::SharedPtr service =
    node->create_service<example_interfaces::srv::AddTwoInts>("add_two_ints", &add);

  RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "Ready to add two ints.");

  rclcpp::spin(node);
  rclcpp::shutdown();
}
```

Add executable: Agregar el siguiente código al CMakeLists.txt del paquete:

```
add_executable(server src/add_two_ints_server.cpp)
ament_target_dependencies(server
rclcpp example_interfaces)
```

Para que ros2 run pueda encontrar el ejecutable, se agrega lo siguiente al final del archivo CMakeLists.txt:

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

Ahora se debe escribir el nodo del cliente, dentro de src del paquete crear un archivo llamado *add_two_ints_client.cpp* y copiar lo siguiente:

```
#include "rclcpp/rclcpp.hpp"
#include "example_interfaces/srv/add_two_ints.hpp"

#include <chrono>
#include <cstdlib>
#include <memory>

using namespace std::chrono_literals;

int main(int argc, char **argv)
{
  rclcpp::init(argc, argv);

  if (argc != 3) {
      RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "usage: add_two_ints_client X Y");
      return 1;
  }

  std::shared_ptr<rclcpp::Node> node = rclcpp::Node::make_shared("add_two_ints_client");
  rclcpp::Client<example_interfaces::srv::AddTwoInts>::SharedPtr client =
    node->create_client<example_interfaces::srv::AddTwoInts>("add_two_ints");

  auto request = std::make_shared<example_interfaces::srv::AddTwoInts::Request>();
  request->a = atoll(argv[1]);
  request->b = atoll(argv[2]);

  while (!client->wait_for_service(1s)) {
    if (!rclcpp::ok()) {
      RCLCPP_ERROR(rclcpp::get_logger("rclcpp"), "Interrupted while waiting for the service.
Exiting.");
      return 0;
    }
    RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "service not available, waiting again...");
  }
```

```
  auto result = client->async_send_request(request);
  // Wait for the result.
  if (rclcpp::spin_until_future_complete(node, result) ==
    rclcpp::FutureReturnCode::SUCCESS)
  {
    RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "Sum: %ld", result.get()->sum);
  } else {
    RCLCPP_ERROR(rclcpp::get_logger("rclcpp"), "Failed to call service add_two_ints");
  }

  rclcpp::shutdown();
  return 0;
}
```

Build and run:

Es una buena práctica correr *rosdep* en la raíz del ws para verificar las dependencias antes de hacer el build:

- rosdep install -i --from-path src --rosdistro foxy -y

Ahora sí se puede proceder con el build del ws.

Creating custom msg and srv files

msg definition:

Para la creación de los archivos .msg y .srv es necesario tener un paquete que los contenga, idealmente un paquete por cada uno. Se crea el paquete:

- ros2 pkg create --build-type ament_cmake tutorial_interfaces

Dentro de la carpeta principal del paquete creado, se deben crear dos carpetas nuevas, una para msg y otra para srv

- mkdir msg
- mkdir srv

Dentro de la carpeta msg crear un archivo llamado *Num.msg* en donde la única información dentro del archivo será:

- *int64 num*

Esto crea un mensaje con un único int de 64 bits llamado num.

Dentro de la misma carpeta /msg crear un archivo llamado *Sphere.msg* que contenga la siguiente información:

- geometry_msgs/Point center
- float64 radius

Este mensaje emplea “mensajes” de otro paquete (geometry_msg/Point)

srv definition:

Dentro de la carpeta srv crear un archivo llamado *AddThreeInts.srv* que contenla la siguiente info:

```
int64 a
int64 b
int64 c
---
int64 sum
```

Este servicio solicita tres int (a, b & c) y responde con un int llamado sum.

CMakeLists.txt: Para convertir las interfaces definidas previamente a un código de lenguaje específico (como C++ o Python) y así poderlas usar, se agregan las siguientes líneas al archivo CMakeLists.txt

```
find_package(geometry_msgs REQUIRED)
find_package(rosidl_default_generators REQUIRED)

rosidl_generate_interfaces(${PROJECT_NAME}
  "msg/Num.msg"
  "msg/Sphere.msg"
  "srv/AddThreeInts.srv"
  DEPENDENCIES geometry_msgs # Add packages that above messages depend on, in this case geometry_msgs for Sphere.msg
)
```

Package.xml:

Se debe añadir al package.xml lo siguiente:

```
<depend>geometry_msgs</depend>

<build_depend>rosidl_default_generators</build_depend>

<exec_depend>rosidl_default_runtime</exec_depend>

<member_of_group>rosidl_interface_packages</member_of_group>
```

Porque la interfaz depende de *rosidel_default_generators* para generar el lenguaje específico del código, se necesitan declarar las dependencias de ella. <*exec_depend*> se usa para especificar el tiempo de ejecución o las dependencias de la etapa de ejecución y *rosidl_interface_packages* es el nombre del grupo de dependencias al cual pertenece el paquete declarado usando el tag <*member_of_group*>.

Ahora se debe realizar un build en la carpeta raíz del ws.

Confirm msg and srv creation:

En una nueva terminal hacer un source local del ws y luego ejecutar:

- ros2 interface show tutorial_interfaces/msg/Num

Se debe imprimir en pantalla:
int64 num

Ahora si se ejecuta:

- ros2 interface show tutorial_interfaces/msg/Sphere

Se debe imprimir en pantalla

```
geometry_msgs/Point center
        float64 x
        float64 y
        float64 z
float64 radius
```

Si se ejecuta:

- ros2 interface show tutorial_interfaces/srv/AddThreeInts

Se imprime en pantalla:

```
int64 a
int64 b
int64 c
---
int64 sum
```

Testing Num.msg with pub/sub:

Con unas pocas modificaciones al paquete cpp_pubsub se puede ver al archivo *Num.msg* en acción.

Los cambios del publisher:

```cpp
#include <chrono>
#include <memory>

#include "rclcpp/rclcpp.hpp"
//#include "std_msgs/msg/string.hpp"
#include "tutorial_interfaces/msg/num.hpp" // CHANGE

using namespace std::chrono_literals;

/* This example creates a subclass of Node and uses std::bind() to register a
* member function as a callback from the timer. */

class MinimalPublisher : public rclcpp::Node
{
public:
MinimalPublisher()
: Node("minimal_publisher"), count_(0)
{
//publisher_ = this->create_publisher<std_msgs::msg::String>("topic", 10);
publisher_ = this->create_publisher<tutorial_interfaces::msg::Num>("topic", 10); // CHANGE
para emplear el msg definido en el paquete creado tutorial_interfaces
timer_ = this->create_wall_timer(
500ms, std::bind(&MinimalPublisher::timer_callback, this));
}

private:
void timer_callback()
{
//auto message = std_msgs::msg::String();
auto message = tutorial_interfaces::msg::Num(); // CHANGE para emplear el msg definido en
el paquete creado tutorial_interfaces
//message.data = "Miguel sexy! " + std::to_string(count_++);
message.num = this->count_++; // CHANGE para emplear el msg definido en el paquete
creado tutorial_interfaces
//RCLCPP_INFO(this->get_logger(), "Publishing: '%s'", message.data.c_str());
RCLCPP_INFO(this->get_logger(), "Publishing: '%d'", message.num); // CHANGE para emplear
el msg definido en el paquete creado tutorial_interfaces
publisher_->publish(message);
}
rclcpp::TimerBase::SharedPtr timer_;
//rclcpp::Publisher<std_msgs::msg::String>::SharedPtr publisher_;
rclcpp::Publisher<tutorial_interfaces::msg::Num>::SharedPtr publisher_; // CHANGE para
emplear el msg definido en el paquete creado tutorial_interfaces
size_t count_;
};

int main(int argc, char * argv[])
```

```
{
rclcpp::init(argc, argv);
rclcpp::spin(std::make_shared<MinimalPublisher>());
rclcpp::shutdown();
return 0;
}
```

Los cambios del subscriber:

```
#include <memory>

#include "rclcpp/rclcpp.hpp"
//#include "std_msgs/msg/string.hpp"
#include "tutorial_interfaces/msg/num.hpp" // CHANGE para usar tutorial_interfaces
using std::placeholders::_1;

class MinimalSubscriber : public rclcpp::Node
{
public:
MinimalSubscriber()
: Node("minimal_subscriber")
{
//subscription_ = this->create_subscription<std_msgs::msg::String>(
subscription_ = this->create_subscription<tutorial_interfaces::msg::Num>( // CHANGE para
usar tutorial_interfaces
"topic", 10, std::bind(&MinimalSubscriber::topic_callback, this, _1));
}

private:
//void topic_callback(const std_msgs::msg::String::SharedPtr msg) const
void topic_callback(const tutorial_interfaces::msg::Num::SharedPtr msg) const // CHANGE
para usar tutorial_interfaces
{
//RCLCPP_INFO(this->get_logger(), "I heard: '%s'", msg->data.c_str());
RCLCPP_INFO(this->get_logger(), "I heard: '%d'", msg->num); // CHANGE para usar
tutorial_interfaces
}
//rclcpp::Subscription<std_msgs::msg::String>::SharedPtr subscription_;
rclcpp::Subscription<tutorial_interfaces::msg::Num>::SharedPtr subscription_; // CHANGE
para usar tutorial_interfaces
};

int main(int argc, char * argv[])
{
```

```
rclcpp::init(argc, argv);
rclcpp::spin(std::make_shared<MinimalSubscriber>());
rclcpp::shutdown();
return 0;
}
```

Y el CMakeLists

```
cmake_minimum_required(VERSION 3.5)
project(cpp_pubsub)

# Default to C99
if(NOT CMAKE_C_STANDARD)
set(CMAKE_C_STANDARD 99)
endif()

# Default to C++14
if(NOT CMAKE_CXX_STANDARD)
set(CMAKE_CXX_STANDARD 14)
endif()

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(rclcpp REQUIRED)
#find_package(std_msgs REQUIRED)
find_package(tutorial_interfaces REQUIRED) # CHANGE para usar tutorial_interfaces

add_executable(talker src/publisher_member_function.cpp)
#ament_target_dependencies(talker rclcpp std_msgs)
ament_target_dependencies(talker rclcpp tutorial_interfaces) # CHANGE para usar
tutorial_interfaces

add_executable(listener src/subscriber_member_function.cpp)
#ament_target_dependencies(listener rclcpp std_msgs)
ament_target_dependencies(listener rclcpp tutorial_interfaces) # CHANGE para usar
tutorial_interfaces

install(TARGETS
talker
listener
DESTINATION lib/${PROJECT_NAME})

if(BUILD_TESTING)
```

```
find_package(ament_lint_auto REQUIRED)
ament_lint_auto_find_test_dependencies()
endif()

ament_package()
```

En el archivo package.xml agregar la siguiente línea:

- <depend>tutorial_interfaces</depend>

Ahora se debe hacer un build para poder usar los paquetes.

Abrir dos teminales y hacer source en cada una, luego ejecutar le talker y el publisher que se crearon anteriormente en el paquete cpp_pubsub.

Testing AddThreeInts.srv with service/client:

Con algunas modificaciones sobre el paquete service/client creado anteriormente, se puede ver a AddThreeInts.srv en acción.

Server node:

```
#include "rclcpp/rclcpp.hpp"
//#include "example_interfaces/srv/add_two_ints.hpp"
#include "tutorial_interfaces/srv/add_three_ints.hpp" // CHANGE

#include <memory>

//void add(const std::shared_ptr<example_interfaces::srv::AddTwoInts::Request> request,
void add(const std::shared_ptr<tutorial_interfaces::srv::AddThreeInts::Request> request, //
CHANGE
//std::shared_ptr<example_interfaces::srv::AddTwoInts::Response> response)
std::shared_ptr<tutorial_interfaces::srv::AddThreeInts::Response> response) // CHANGE
{
//response->sum = request->a + request->b;
response->sum = request->a + request->b + request->c; // CHANGE
//RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "Incoming request\na: %ld" " b: %ld",
// request->a, request->b);
RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "Incoming request\na: %ld" " b: %ld" " c: %ld", //
CHANGE
request->a, request->b, request->c); // CHANGE
RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "sending back response: [%ld]", (long int)response-
>sum);
}
```

```
int main(int argc, char **argv)
{
rclcpp::init(argc, argv);

//std::shared_ptr<rclcpp::Node> node = rclcpp::Node::make_shared("add_two_ints_server");
std::shared_ptr<rclcpp::Node> node =
rclcpp::Node::make_shared("add_three_ints_server"); // CHANGE


//rclcpp::Service<example_interfaces::srv::AddTwoInts>::SharedPtr service =
//node->create_service<example_interfaces::srv::AddTwoInts>("add_two_ints", &add);
rclcpp::Service<tutorial_interfaces::srv::AddThreeInts>::SharedPtr service = // CHANGE
node->create_service<tutorial_interfaces::srv::AddThreeInts>("add_three_ints", &add); //
CHANGE

//RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "Ready to add two ints.");
RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "Ready to suck Miguel's dick uwu."); // CHANGE

rclcpp::spin(node);
rclcpp::shutdown();
}
```

Client node:

```
#include "rclcpp/rclcpp.hpp"
//#include "example_interfaces/srv/add_two_ints.hpp"
#include "tutorial_interfaces/srv/add_three_ints.hpp" // CHANGE

#include <chrono>
#include <cstdlib>
#include <memory>

using namespace std::chrono_literals;

int main(int argc, char **argv)
{
rclcpp::init(argc, argv);
/*
if (argc != 3) {
RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "usage: add_two_ints_client X Y");
return 1;
}
*/
if (argc != 4) { // CHANGE
RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "usage: add_three_ints_client X Y Z"); // CHANGE
```

```
return 1;
}

//std::shared_ptr<rclcpp::Node> node = rclcpp::Node::make_shared("add_two_ints_client");
std::shared_ptr<rclcpp::Node> node = rclcpp::Node::make_shared("add_three_ints_client"); //
CHANGE
//rclcpp::Client<example_interfaces::srv::AddTwoInts>::SharedPtr client =
//node->create_client<example_interfaces::srv::AddTwoInts>("add_two_ints");
rclcpp::Client<tutorial_interfaces::srv::AddThreeInts>::SharedPtr client = // CHANGE
node->create_client<tutorial_interfaces::srv::AddThreeInts>("add_three_ints"); // CHANGE

auto request = std::make_shared<tutorial_interfaces::srv::AddThreeInts::Request>();
request->a = atoll(argv[1]);
request->b = atoll(argv[2]);
request->c = atoll(argv[3]); // CHANGE

while (!client->wait_for_service(1s)) {
if (!rclcpp::ok()) {
RCLCPP_ERROR(rclcpp::get_logger("rclcpp"), "Interrupted while waiting for the service.
Exiting.");
return 0;
}
RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "service not available, waiting again...");
}

auto result = client->async_send_request(request);
// Wait for the result.
if (rclcpp::spin_until_future_complete(node, result) ==
rclcpp::FutureReturnCode::SUCCESS)
{
RCLCPP_INFO(rclcpp::get_logger("rclcpp"), "Sum: %ld", result.get()->sum);
} else {
//RCLCPP_ERROR(rclcpp::get_logger("rclcpp"), "Failed to call service add_two_ints");
RCLCPP_ERROR(rclcpp::get_logger("rclcpp"), "Failed to call service add_three_ints"); //
CHANGE
}

rclcpp::shutdown();
return 0;
}
```

Luego se añade al CMakeLists:

```
#...

# find dependencies
```

```
find_package(ament_cmake REQUIRED)
find_package(rclcpp REQUIRED)
#find_package(example_interfaces REQUIRED)
find_package(tutorial_interfaces REQUIRED) # CHANGE

add_executable(server src/add_two_ints_server.cpp)
ament_target_dependencies(server
#rclcpp example_interfaces)
rclcpp tutorial_interfaces) #CHANGE


add_executable(client src/add_two_ints_client.cpp)
ament_target_dependencies(client
#rclcpp example_interfaces)
rclcpp tutorial_interfaces) #CHANGE

install(TARGETS
server
client
DESTINATION lib/${PROJECT_NAME})

if(BUILD_TESTING)
find_package(ament_lint_auto REQUIRED)
ament_lint_auto_find_test_dependencies()
endif()

ament_package()
```

Por último, al package.xml:

```
<depend>tutorial_interfaces</depend>
```

Se realiza el build en la raíz del ws.

En dos terminales diferentes, hacer un source local y ejecutar:

- ros2 run cpp_srvcli server
- ros2 run cpp_srvcli client 2 3 1

Managing dependencies with rosdep

what is rosdep?: rosdep es la utilidad de ros para administrar dependencias que pueden trabajar con los paquetes y librerías externas. Rosdep es una línea de comando para identificar e instalar dependencias para realizar un build o para instalar paquetes.

Rosdep hace una revisión de los archivos *package.xml* en el ws o en un paquete específico y encuentra las *rosdep keys* guardadas en él. Estas *keys* se comparan con el índice central de ROS central para encontrar los paquetes o las librerías en varios administradores de paquetes, finalmente, cuando encuentre los paquetes y librerías, estas serán instaladas.
El índice central se conoce como *rosdistro*.

¿Cómo uso la herramienta de rosdep: Si es la primera vez usando rosdep, se tiene que inicializar primero:

- sudo rosdep init
- rosdep update

Ahora se podrá correr la instalación:

- rosdep install --from-paths src -y –ignore-src

--from-paths especifica la ruta a chequear para el *package.xml* y resolver las *keys*.
-y significa “sí”, es decir, da vía libre para que se instale todo
--ignore-src es para ignorar las dependencia.

Creating an action:

Para el tutorial, se crea un paquete dentro del ws llamado *action_tutorials_interfaces*

- *ros2 pkg create --build-type ament_cmake action_tutorials_interfaces*

Las acciones están definidas por archivos *.action* que tienen la forma:

```
# Request
---
# Result
---
# Feedback
```

Un mensaje de *request* es enviado desde un *action client* hace un *action server* iniciando una nueva meta.

Un mensaje de *result* es enviado desde un *action server* hacia un *action client* cuando la meta está completa.

El *feedback* son mensajes enviados periódicamente desde un *action server* hacia un *action client* con actualizaciones acerca de la meta fijada.

Continuando con el tutorial, se realizará una secuancia Fibonacci, para eso, se crea una carpeta llamada “action” dentro del paquete creado.

- Cd action_tutorials_interfaces
- mkdir action

Dentro de la carpeta “action” crear un archivo llamado *Fibonacci.action* que contenga lo siguiente:

```
int32 order
---
int32[] sequence
---
int32[] partial_sequence
```

La meta es el orden (order) de la secuecia Fibonacci que se quiere computar, el resultado es el final de la secuencia (sequence), y la retroalimentación (feedback) es la secuencia parcial.

Building an action: Antes de poder usar la acción, se debe pasar la definición a rosidl. Dentro del archivo CMakeLists.txt del paquete, agregar las siguientes líneas antes del *ament_package()*.

```
find_package(rosidl_default_generators REQUIRED)

rosidl_generate_interfaces(${PROJECT_NAME}
  "action/Fibonacci.action"
)
```

También se deben agregar las dependencias requeridas en el archivo *package.xml*.

```
<buildtool_depend>rosidl_default_generators</buildtool_depend>

<depend>action_msgs</depend>

<member_of_group>rosidl_interface_packages</member_of_group>
```

Ahora sí se podrá hacer el build del paquete; dirigirse a la raíz del ws y ejecutar *colcon build*.

Por convención, las acciones tienen como prefijo wl nombre del paquete y la palabra action, entonces, si nos queremos referir a la acción creada, será de la forma *action_tutorials_interfaces/action/Fibonacci*.

En este tutorial se aprendió sobre la estructura para la definición de una acción y como contruir una.

Writing an action server and client (C++)

Las acciones son una forma de comunicación asíncrona en ROS. Los clientes envían una meta y los servidores envían el resultado y una retroalimentación del estado del proceso hasta alcanzar la meta.

Creamos el paquete:

- cd ~/ros2_ws/src
- ros2 pkg create --dependencies action_tutorials_interfaces rclcpp rclcpp_action rclcpp_components – action_tutorials_cpp

Dentro del directorio *action_tutorials_cpp/include/action_tutorials_cpp* crear un archivo llamado *visibility_control.h* y pegue el siguiente código:

```
#ifndef ACTION_TUTORIALS_CPP__VISIBILITY_CONTROL_H_
#define ACTION_TUTORIALS_CPP__VISIBILITY_CONTROL_H_

#ifdef __cplusplus
extern "C"
{
#endif

// This logic was borrowed (then namespaced) from the examples on the gcc wiki:
// https://gcc.gnu.org/wiki/Visibility

#if defined _WIN32 || defined __CYGWIN__
#ifdef __GNUC__
#define ACTION_TUTORIALS_CPP_EXPORT __attribute__ ((dllexport))
#define ACTION_TUTORIALS_CPP_IMPORT __attribute__ ((dllimport))
#else
#define ACTION_TUTORIALS_CPP_EXPORT __declspec(dllexport)
#define ACTION_TUTORIALS_CPP_IMPORT __declspec(dllimport)
#endif
#ifdef ACTION_TUTORIALS_CPP_BUILDING_DLL
#define ACTION_TUTORIALS_CPP_PUBLIC ACTION_TUTORIALS_CPP_EXPORT
#else
#define ACTION_TUTORIALS_CPP_PUBLIC ACTION_TUTORIALS_CPP_IMPORT
#endif
#define ACTION_TUTORIALS_CPP_PUBLIC_TYPE ACTION_TUTORIALS_CPP_PUBLIC
#define ACTION_TUTORIALS_CPP_LOCAL
#else
#define ACTION_TUTORIALS_CPP_EXPORT __attribute__ ((visibility("default")))
#define ACTION_TUTORIALS_CPP_IMPORT
#if __GNUC__ >= 4
#define ACTION_TUTORIALS_CPP_PUBLIC __attribute__ ((visibility("default")))
#define ACTION_TUTORIALS_CPP_LOCAL __attribute__ ((visibility("hidden")))
#else
```

```
#define ACTION_TUTORIALS_CPP_PUBLIC
#define ACTION_TUTORIALS_CPP_LOCAL
#endif
#define ACTION_TUTORIALS_CPP_PUBLIC_TYPE
#endif

#ifdef __cplusplus
}
#endif

#endif // ACTION_TUTORIALS_CPP__VISIBILITY_CONTROL_H_
```

Writing an action server: Se escribirá el nodo servidor para computar la secuencia Fibonacci usando la acción creada anteriormente.

Dentro del directorio *action_tutorials_cpp/src* crear un archivo llamado *fibonacci_action_server.cpp* y pegar el siguiente código:

```
#include <functional>
#include <memory>
#include <thread>

#include "action_tutorials_interfaces/action/fibonacci.hpp"
#include "rclcpp/rclcpp.hpp"
#include "rclcpp_action/rclcpp_action.hpp"
#include "rclcpp_components/register_node_macro.hpp"

#include "action_tutorials_cpp/visibility_control.h"

namespace action_tutorials_cpp
{
class FibonacciActionServer : public rclcpp::Node
{
public:
using Fibonacci = action_tutorials_interfaces::action::Fibonacci;
using GoalHandleFibonacci = rclcpp_action::ServerGoalHandle<Fibonacci>;

ACTION_TUTORIALS_CPP_PUBLIC
explicit FibonacciActionServer(const rclcpp::NodeOptions & options = rclcpp::NodeOptions())
: Node("fibonacci_action_server", options)
{
using namespace std::placeholders;

this->action_server_ = rclcpp_action::create_server<Fibonacci>(
this,
"fibonacci",
std::bind(&FibonacciActionServer::handle_goal, this, _1, _2),
```

```
std::bind(&FibonacciActionServer::handle_cancel, this, _1),
std::bind(&FibonacciActionServer::handle_accepted, this, _1));
}

private:
rclcpp_action::Server<Fibonacci>::SharedPtr action_server_;

rclcpp_action::GoalResponse handle_goal(
const rclcpp_action::GoalUUID & uuid,
std::shared_ptr<const Fibonacci::Goal> goal)
{
RCLCPP_INFO(this->get_logger(), "Received goal request with order %d", goal->order);
(void)uuid;
return rclcpp_action::GoalResponse::ACCEPT_AND_EXECUTE;
}

rclcpp_action::CancelResponse handle_cancel(
const std::shared_ptr<GoalHandleFibonacci> goal_handle)
{
RCLCPP_INFO(this->get_logger(), "Received request to cancel goal");
(void)goal_handle;
return rclcpp_action::CancelResponse::ACCEPT;
}

void handle_accepted(const std::shared_ptr<GoalHandleFibonacci> goal_handle)
{
using namespace std::placeholders;
// this needs to return quickly to avoid blocking the executor, so spin up a new thread
std::thread{std::bind(&FibonacciActionServer::execute, this, _1), goal_handle}.detach();
}

void execute(const std::shared_ptr<GoalHandleFibonacci> goal_handle)
{
RCLCPP_INFO(this->get_logger(), "Executing goal");
rclcpp::Rate loop_rate(1);
const auto goal = goal_handle->get_goal();
auto feedback = std::make_shared<Fibonacci::Feedback>();
auto & sequence = feedback->partial_sequence;
sequence.push_back(0);
sequence.push_back(1);
auto result = std::make_shared<Fibonacci::Result>();

for (int i = 1; (i < goal->order) && rclcpp::ok(); ++i) {
// Check if there is a cancel request
if (goal_handle->is_canceling()) {
result->sequence = sequence;
goal_handle->canceled(result);
```

```
RCLCPP_INFO(this->get_logger(), "Goal canceled");
return;
}
// Update sequence
sequence.push_back(sequence[i] + sequence[i - 1]);
// Publish feedback
goal_handle->publish_feedback(feedback);
RCLCPP_INFO(this->get_logger(), "Publish feedback");

loop_rate.sleep();
}

// Check if goal is done
if (rclcpp::ok()) {
result->sequence = sequence;
goal_handle->succeed(result);
RCLCPP_INFO(this->get_logger(), "Goal succeeded");
}
}
}; // class FibonacciActionServer

} // namespace action_tutorials_cpp

RCLCPP_COMPONENTS_REGISTER_NODE(action_tutorials_cpp::FibonacciActionServer)
```

El código anterior crea una clase llamada *FibonacciActionServer*, y tienen un constructor que se inicializa con el nodo *fibonacci_action_server*. El constructor también tiene:

```
this->action_server_ = rclcpp_action::create_server<Fibonacci>(
  this,
  "fibonacci",
  std::bind(&FibonacciActionServer::handle_goal, this, _1, _2),
  std::bind(&FibonacciActionServer::handle_cancel, this, _1),
  std::bind(&FibonacciActionServer::handle_accepted, this, _1));
```

Un *action server* requiere 6 cosas:

1. El nombre del tipo de acción *(Fibonacci)*.
2. Un nodo de ROS 2 para agregar la acción al *this*.
3. El nombre de la acción *"fibonacci"*.
4. El llamado de la meta: *handle_goal*
5. El llamado de la cancelación: *handle_cancel*
6. El llamado de la aceptación de la meta: *handle_accept*

Compiling the action server: Primero se debe actualizar el archivo CMakeLists.txt agregando el siguiente código luego del llamado *find_package*.

```
add_library(action_server SHARED
  src/fibonacci_action_server.cpp)
target_include_directories(action_server PRIVATE
  $<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}/include>
  $<INSTALL_INTERFACE:include>)
target_compile_definitions(action_server
  PRIVATE "ACTION_TUTORIALS_CPP_BUILDING_DLL")
ament_target_dependencies(action_server
  "action_tutorials_interfaces"
  "rclcpp"
  "rclcpp_action"
  "rclcpp_components")
rclcpp_components_register_node(action_server PLUGIN
"action_tutorials_cpp::FibonacciActionServer" EXECUTABLE fibonacci_action_server)
install(TARGETS
  action_server
  ARCHIVE DESTINATION lib
  LIBRARY DESTINATION lib
  RUNTIME DESTINATION bin)
```

Ahora se puede compilar el paquete con *colcon build*.

Writing an action client: En el directorio action_tutorials_cpp/src crear un archivo llamado *fibonacci_action_client.cpp*. Copiar el siguiente código:

```
#include <functional>
#include <future>
#include <memory>
#include <string>
#include <sstream>

#include "action_tutorials_interfaces/action/fibonacci.hpp"

#include "rclcpp/rclcpp.hpp"
#include "rclcpp_action/rclcpp_action.hpp"
#include "rclcpp_components/register_node_macro.hpp"

namespace action_tutorials_cpp
{
class FibonacciActionClient : public rclcpp::Node
{
public:
  using Fibonacci = action_tutorials_interfaces::action::Fibonacci;
  using GoalHandleFibonacci = rclcpp_action::ClientGoalHandle<Fibonacci>;

  explicit FibonacciActionClient(const rclcpp::NodeOptions & options)
  : Node("fibonacci_action_client", options)
  {
```

```
    this->client_ptr_ = rclcpp_action::create_client<Fibonacci>(
      this,
      "fibonacci");

    this->timer_ = this->create_wall_timer(
      std::chrono::milliseconds(500),
      std::bind(&FibonacciActionClient::send_goal, this));
  }

  void send_goal()
  {
    using namespace std::placeholders;

    this->timer_->cancel();

    if (!this->client_ptr_->wait_for_action_server()) {
      RCLCPP_ERROR(this->get_logger(), "Action server not available after waiting");
      rclcpp::shutdown();
    }

    auto goal_msg = Fibonacci::Goal();
    goal_msg.order = 10;

    RCLCPP_INFO(this->get_logger(), "Sending goal");

    auto send_goal_options = rclcpp_action::Client<Fibonacci>::SendGoalOptions();
    send_goal_options.goal_response_callback =
      std::bind(&FibonacciActionClient::goal_response_callback, this, _1);
    send_goal_options.feedback_callback =
      std::bind(&FibonacciActionClient::feedback_callback, this, _1, _2);
    send_goal_options.result_callback =
      std::bind(&FibonacciActionClient::result_callback, this, _1);
    this->client_ptr_->async_send_goal(goal_msg, send_goal_options);
  }

private:
  rclcpp_action::Client<Fibonacci>::SharedPtr client_ptr_;
  rclcpp::TimerBase::SharedPtr timer_;

  void goal_response_callback(std::shared_future<GoalHandleFibonacci::SharedPtr> future)
  {
    auto goal_handle = future.get();
    if (!goal_handle) {
      RCLCPP_ERROR(this->get_logger(), "Goal was rejected by server");
    } else {
      RCLCPP_INFO(this->get_logger(), "Goal accepted by server, waiting for result");
    }
  }
```

```
  void feedback_callback(
    GoalHandleFibonacci::SharedPtr,
    const std::shared_ptr<const Fibonacci::Feedback> feedback)
  {
    std::stringstream ss;
    ss << "Next number in sequence received: ";
    for (auto number : feedback->partial_sequence) {
      ss << number << " ";
    }
    RCLCPP_INFO(this->get_logger(), ss.str().c_str());
  }

  void result_callback(const GoalHandleFibonacci::WrappedResult & result)
  {
    switch (result.code) {
      case rclcpp_action::ResultCode::SUCCEEDED:
        break;
      case rclcpp_action::ResultCode::ABORTED:
        RCLCPP_ERROR(this->get_logger(), "Goal was aborted");
        return;
      case rclcpp_action::ResultCode::CANCELED:
        RCLCPP_ERROR(this->get_logger(), "Goal was canceled");
        return;
      default:
        RCLCPP_ERROR(this->get_logger(), "Unknown result code");
        return;
    }
    std::stringstream ss;
    ss << "Result received: ";
    for (auto number : result.result->sequence) {
      ss << number << " ";
    }
    RCLCPP_INFO(this->get_logger(), ss.str().c_str());
    rclcpp::shutdown();
  }
};  // class FibonacciActionClient

}  // namespace action_tutorials_cpp

RCLCPP_COMPONENTS_REGISTER_NODE(action_tutorials_cpp::FibonacciActionClient)
```

Para compilar el action client se debe agregar al CMakeLists.txt debajo de *find_packages*:

```
add_library(action_client SHARED
  src/fibonacci_action_client.cpp)
target_include_directories(action_client PRIVATE
  $<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}/include>
  $<INSTALL_INTERFACE:include>)
target_compile_definitions(action_client
```

```
  PRIVATE "ACTION_TUTORIALS_CPP_BUILDING_DLL")
ament_target_dependencies(action_client
  "action_tutorials_interfaces"
  "rclcpp"
  "rclcpp_action"
  "rclcpp_components")
rclcpp_components_register_node(action_client PLUGIN
"action_tutorials_cpp::FibonacciActionClient" EXECUTABLE fibonacci_action_client)
install(TARGETS
  action_client
  ARCHIVE DESTINATION lib
  LIBRARY DESTINATION lib
  RUNTIME DESTINATION bin)
```

realizar el build y ya se puede ejecutar.

Creating a launch file:

El sistema de *launch* en ROS 2 es responsable de ayudar al usuario a describir la configuración de los sistemas y luego ejecutarlos como se describieron. La configuración del sistema incluye que programas correr, donde correrlos, qué argumentos pasarles y las convenciones de ROS para reusar los componentes a través del sistema.
También es responsable del monitoreo del estado del sistema que fue lanzado.

Write the launch file: Ahora se hará uso de turtlesim y sus ejecutables para la creación del *launch file*. Estos archivos pueden ser creados en Python, XML y YAML.
Se hará en Python.

Crear una carpeta llamada launch y dentro de ella un archivo llamado *turtlesim_mimic_launch.py* que contenga el siguiente código:

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

def generate_launch_description():
    return LaunchDescription([
        Node(
            package='turtlesim',
            namespace='turtlesim1',
            executable='turtlesim_node',
            name='sim'
        ),
        Node(
            package='turtlesim',
            namespace='turtlesim2',
```

```
            executable='turtlesim_node',
            name='sim'
        ),
        Node(
            package='turtlesim',
            executable='mimic',
            name='mimic',
            remappings=[
                ('/input/pose', '/turtlesim1/turtle1/pose'),
                ('/output/cmd_vel', '/turtlesim2/turtle1/cmd_vel'),
            ]
        )
    ])
```

Ahora dentro de la carpeta launch se ejecuta:

- ros2 launch turtlesim_mimic_launch.py

Es posible ejecutar un *launch file* directamente como se acabó de ver, o también si el archivo .launch se encuentra en un paquete, si el caso es este último, se ejecuta:

- ros2 launch <package_name> <launch_file_name>

Para los paquetes con *launch files* es bueno agregar *exec_depend* en el archivo *package.xml*:

- <exec_depend>ros2launch</exec_depend>

Esto asegura que el comando *ros2 launch* estará disponible luego de hacer el build del paquete.

Volviendo al ejemplo con el turtlesim, al ejecutar el *launch file* se abrirán dos ventanas con tortugas, el nodo de *mimic* declarado en el *launch file* hace referencia a que el movimiento será el mismo para ambas, es decir, si se ejecuta en otra terminal el nodo del *teleop_turtle* se moverán ambas al tiempo.

Integrating launch files into ros 2 packages:

Anteriormente se vio cómo declara *launch files*, ahora se integrarán a un paquete de un ws.

Dentro de un ws crear un paquete:

- ros2 pkg create cpp_launch_example --build-type ament_cmake

Creating the structure to hold launch files: Por convención, todos los *launch files* de un paquete se guardan dentro de una carpeta llamada *launch*.

Para los paquetes creados con C++ basta ajustar el archivo CMakeLists.txt agregando:

```
# Install launch files.
install(DIRECTORY
  launch
  DESTINATION share/${PROJECT_NAME}/
)
```

Writing a launch file: Dentro de la carpeta launch del paquete, crear un archivo llamado *my_script_launch.py*. Todos los archivos .launch creados en Python deben finalizar con _launch.py.

Dentro del archivo creado, pegar:

```
import launch
import launch_ros.actions

def generate_launch_description():
    return launch.LaunchDescription([
        launch_ros.actions.Node(
            package='demo_nodes_cpp',
            executable='talker',
            name='talker'),
  ])
```

Se debe hacer el build del paquete, y ya estará listo para ejecutar:

- ros2 launch cpp_launch_example my_script_launch.py

Managing large projects:

Se empezará creando un paquete con muchos nodos del turtlesim para aprender a administrarlos:

- ros2 pkg create launch_tutorial --build-type ament_python

Dentro del paquete crear una carpeta llamada *launch* y crera un archivo llamado launch_turtlesim.launch.py donde se pegará el siguiente código:

```
import os

from ament_index_python.packages import get_package_share_directory

from launch import LaunchDescription
from launch.actions import IncludeLaunchDescription
from launch.launch_description_sources import PythonLaunchDescriptionSource
```

```
def generate_launch_description():
  turtlesim_world_1 = IncludeLaunchDescription(
    PythonLaunchDescriptionSource([os.path.join(
      get_package_share_directory('launch_tutorial'), 'launch'),
      '/turtlesim_world_1.launch.py'])
    )
  turtlesim_world_2 = IncludeLaunchDescription(
    PythonLaunchDescriptionSource([os.path.join(
      get_package_share_directory('launch_tutorial'), 'launch'),
      '/turtlesim_world_2.launch.py'])
    )
  broadcaster_listener_nodes = IncludeLaunchDescription(
    PythonLaunchDescriptionSource([os.path.join(
      get_package_share_directory('launch_tutorial'), 'launch'),
      '/broadcaster_listener.launch.py']),
    launch_arguments={'target_frame': 'carrot1'}.items(),
    )
  mimic_node = IncludeLaunchDescription(
    PythonLaunchDescriptionSource([os.path.join(
      get_package_share_directory('launch_tutorial'), 'launch'),
      '/mimic.launch.py'])
    )
  fixed_frame_node = IncludeLaunchDescription(
    PythonLaunchDescriptionSource([os.path.join(
      get_package_share_directory('launch_tutorial'), 'launch'),
      '/fixed_broadcaster.launch.py'])
    )
  rviz_node = IncludeLaunchDescription(
    PythonLaunchDescriptionSource([os.path.join(
      get_package_share_directory('launch_tutorial'), 'launch'),
      '/turtlesim_rviz.launch.py'])
    )

  return LaunchDescription([
    turtlesim_world_1,
    turtlesim_world_2,
    broadcaster_listener_nodes,
    mimic_node,
    fixed_frame_node,
    rviz_node
  ])
```

Setting parameters in the launch file: Ahora se deben crear todos los *launch files* declarados en el archivo padre (el creado anteriormente). Crear un archivo llamado *turtlesim_world_1.launch.py* y pegar el siguiente código:

```
from launch import LaunchDescription
from launch.actions import DeclareLaunchArgument
```

```
from launch.substitutions import LaunchConfiguration, TextSubstitution

from launch_ros.actions import Node


def generate_launch_description():
  background_r_launch_arg = DeclareLaunchArgument(
    'background_r', default_value=TextSubstitution(text='0')
  )
  background_g_launch_arg = DeclareLaunchArgument(
    'background_g', default_value=TextSubstitution(text='84')
  )
  background_b_launch_arg = DeclareLaunchArgument(
    'background_b', default_value=TextSubstitution(text='122')
  )

  return LaunchDescription([
    background_r_launch_arg,
    background_g_launch_arg,
    background_b_launch_arg,
    Node(
      package='turtlesim',
      executable='turtlesim_node',
      name='sim',
      parameters=[{
        'background_r': LaunchConfiguration('background_r'),
        'background_g': LaunchConfiguration('background_g'),
        'background_b': LaunchConfiguration('background_b'),
      }]
    ),
  ])
```

Loading parameters from a YAML file: En el segundo *launch file* (sin contar el padre) se creará un archivo llamado *turtlesim_world_2.launch.py* y se pegará el siguiente código

```
import os

from ament_index_python.packages import get_package_share_directory

from launch import LaunchDescription
from launch_ros.actions import Node


def generate_launch_description():
  config = os.path.join(
    get_package_share_directory('launch_tutorial'),
    'config',
    'turtlesim.yaml'
```

```
    )

  return LaunchDescription([
    Node(
      package='turtlesim',
      executable='turtlesim_node',
      namespace='turtlesim2',
      name='sim',
      parameters=[config]
    )
  ])
```

Este *launch file* "lanzará" el mismo nodo de *turtleism_node* pero los parámetros tendrán los valores cargados directamente de la configuración del archivo YAML. Definir los argumentos y parámetros en archivos YAML hace que sea mucho más sencillo guardar y cargar un gran número de variables.

Ahora crearemos el archivo .YAML que será cargado en el launch definido; para eso, se creará una carpeta dentro del paquete llamada *config* y dentro de esta un archivo llamado *turtlesim.yaml* al cual se le pegará el siguiente código:

```
/turtlesim2/sim:
  ros__parameters:
    background_b: 255
    background_g: 86
    background_r: 150
```

Ahora si se empieza *turtleism_world_2.launch.py*, se empezará a ejecutar el *turtleism_node* con las preconfiguraciones de *background_colors*.

Using woldcard in YAML files: Hay casos en los que se quieren establecer los mismos parámetros en más de un nodo, estos nodos pueden tener diferentes nombres pero pueden requerir los mismos parámetros, definirlo en un archivo YAML por separado para cada nodo no es eficiente; una solución es usar *wildcard characters*, que son una sustitución para caracteres desconocidos.

Se creará un nuevo *launch file* similar al *turtlesim_world_2.launch.py llamado* turtlesim_world3.*launch.py, pero la definición del nodo será ligeramente diferente, lo único que cambia es:*

```
...
Node(
  package='turtlesim',
  executable='turtlesim_node',
  namespace='turtlesim3',
  name='sim',
  parameters=[config]
)
```

Para cargar los parámetros, no es necesario crear un nuevo archivo .YAML, por el contrario, se puede modificar el anterior:

```
/**:
  ros__parameters:
    background_b: 255
    background_g: 86
    background_r: 150
```

La sintaxis del *wildcard* es /**: lo cual le indica al programa que no importa el nombre del nodo, y que cualquier nodo con los mismos parámetros puede cargar los valores desde el .yaml.

Namespaces: Los namespaces permiten al sistema empezar dor nodos similares sin conflicto con los nombres o los topics.

Reusing nodes: Se creará un archivo llamado *broadcaster_listener.launch.py* y se pondrá el siguiente código:

```
from launch import LaunchDescription
from launch.actions import DeclareLaunchArgument
from launch.substitutions import LaunchConfiguration

from launch_ros.actions import Node


def generate_launch_description():
  return LaunchDescription([
    DeclareLaunchArgument(
      'target_frame', default_value='turtle1',
      description='Target frame name.'
    ),
    Node(
      package='turtle_tf2_py',
      executable='turtle_tf2_broadcaster',
      name='broadcaster1',
      parameters=[
        {'turtlename': 'turtle1'}
      ]
    ),
    Node(
      package='turtle_tf2_py',
      executable='turtle_tf2_broadcaster',
      name='broadcaster2',
      parameters=[
        {'turtlename': 'turtle2'}
      ]
    ),
    Node(
      package='turtle_tf2_py',
      executable='turtle_tf2_listener',
      name='listener',
      parameters=[
```

```
            {'target_frame': LaunchConfiguration('target_frame')}
        ]
    ),
  ])
```

Remapping: Ahora cree un archivo llamado *mimic.launch.py* y pegue el siguiente código:

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


def generate_launch_description():
  return LaunchDescription([
    Node(
      package='turtlesim',
      executable='mimic',
      name='mimic',
      remappings=[
        ('/input/pose', '/turtle2/pose'),
        ('/output/cmd_vel', '/turtlesim2/turtle1/cmd_vel'),
      ]
    )
  ])
```

Este archivo empieza el nodo *mimic* el cual le da comandos a una tortuga para que siga a la otra. El nodo está designado para recibir el objetivo de la posición del topic */turtle/pose*. Finalmente, se remapea la salida */output/cmd_vel* cambiándola por */turtlesim2/turtle1/cmd_vel*, así, la tortuga 1 en el mundo turtlesim2 segiurá a la tortuga 2 del mundo inicial.

Config files: Ahora se crea un archivo llamado *turtlesim_rviz.launch.py* y pegar el siguiente código:

```
import os

from ament_index_python.packages import get_package_share_directory

from launch import LaunchDescription
from launch_ros.actions import Node


def generate_launch_description():
  rviz_config = os.path.join(
    get_package_share_directory('turtle_tf2_py'),
    'rviz',
    'turtle_rviz.rviz'
    )

  return LaunchDescription([
    Node(
```

```
        package='rviz2',
        executable='rviz2',
        name='rviz2',
        arguments=['-d', rviz_config]
    )
  ])
```

Este *launch file* empezará Rviz con la configuración definida en el paquete *turtle_tf2_py*.

Enviroment variables: Ahora se creará un archivo llamado *fixed_broadcaster.launch.py* dentro del paquete y se pegará el siguiente código:

```
from launch import LaunchDescription
from launch.actions import DeclareLaunchArgument
from launch.substitutions import EnvironmentVariable, LaunchConfiguration
from launch_ros.actions import Node


def generate_launch_description():
  return LaunchDescription([
    DeclareLaunchArgument(
        'node_prefix',
        default_value=[EnvironmentVariable('USER'), '_'],
        description='prefix for node name'
    ),
    Node(
        package='turtle_tf2_py',
        executable='fixed_frame_tf2_broadcaster',
        name=[LaunchConfiguration('node_prefix'), 'fixed_broadcaster'],
    ),
  ])
```

Este archivo muestra la forma en la cual las *enviromental variables* pueden ser llamadas dentro de los *launch files*. Las *enviromental variables* pueden ser usadas para definir *namespaces* para distinguir los nodos en diferentes robots.

Runing launch files: Primero se tiene que agregar al archivo *setup.py* la siguiente información para que se puedan instalar los *launch files*:

```
data_files=[
    ...
    (os.path.join('share', package_name, 'launch'),
      glob(os.path.join('launch', '*.launch.py'))),
    (os.path.join('share', package_name, 'config'),
      glob(os.path.join('config', '*.yaml'))),
  ],
```

Build and run: Para ver los resultados del código, se tiene que realizar el source y el build del ws, y ejecutar:

- ros2 launch launch_tutorial launch_turtlesim.launch.py

tf

Inicialmente, se debe instalar la demo:

- sudo apt-get install ros-foxy-turtle-tf2-py ros-foxy-tf2-tools ros-foxy-tf-transformations

Ahora se puede ejecutar la demo; en una nueva terminal ejecutar:

- ros2 launch turtle_tf2_py turtle_tf2_demo.launch.py

En otra terminal correr el nodo del teleop_turtle:

- ros2 run turtlesim turtle_teleop_key

Se podrá notar que la tortuga 2 sigue la trayectoria de la torguta 1, pero ¿Qué está pasando?
Esta demo está usando la librería tf2 para crear tres sistemas coordenados. Este tutorial usa el *tf2 broadcaster* para publicar las coordenadas y el *tf2 listener* para computar la diferencia en los sistemas de coordenadas de las tortugas y mover una respecto a la otra.

Una herramienta de t2f es el *tf2_echo* que reporta las transformaciones entre dos sistemas.

- ros2 run tf2_ros tf2_echo [reference_frame] [target_frame]

Si se emplea para mirar las transformaciones entre turtle1 y turtle2:

- ros2 run tf2_ros tf2_echo turtle2 turtle1

rviz and tf2: Rviz es una herramienta de visualización que es útil para examinar los marcos del tf2. Iniciaremos el rviz usando la configuración -d:

- ros2 run rviz2 rviz2 -d $(ros2 pkg prefix --share turtle_tf2_py)/rviz/turtle_rviz.rviz

Writing a static broadcaster (C++): Publicar transformaciones estáticas es útil para definir las relaciones entre la base de un robot y sus sensores o partes no móviles.

Primero se creará un paquete que será usado para este tutorial y los siguientes, el paquete se llamará *learning_tf2_cpp* y tendrá como dependencias *geometry__msgs, rclcpp, tf2, tf2_ros & turtlesim*.

- ros2 pkg create --build-type ament_cmake --dependencies geometry_msgs rclcpp tf2 tf2_ros turtlesim – learning_tf2_cpp

Ahora, dentro de la carpeta src del paquete, digitar en consola:

- wget https://raw.githubusercontent.com/ros/geometry_tutorials/ros2/turtle_tf2_cpp/src/static_turtle_tf2_broadcaster.cpp

Esto creará un archivo llamado *static_turtle_tf2_broadcaster.cpp*.
.Add dependencies: Se tiene que navegar un nivel atrás, hasta la carpeta raíz del PAQUETE *...src/learning_tf2_cpp* donde se encuentra el archivo *CMakeLists.txt* y *package.xml* en el archivo .xml no olvodar diligenciar los campos de *descripiton, maintainer& license*. En el *CMakeLists.txt* se deben agregar los ejecutables:

```
add_executable(static_turtle_tf2_broadcaster src/static_turtle_tf2_broadcaster.cpp)
ament_target_dependencies(
  static_turtle_tf2_broadcaster
  geometry_msgs
  rclcpp
  tf2
  tf2_ros
)
```

Luego dentro del mismo archivo se deben agregar los *install targets*:

```
install(TARGETS
  static_turtle_tf2_broadcaster
  DESTINATION lib/${PROJECT_NAME})
```

Verificar las rosdep escribiendo el sigiente código en la terminal desde la raíz del ws:

- rosdep install -i --from-path src --rosdistro foxy -y

Realizar el build del paquete:

- colcon build --packages-select learning_tf2_cpp

Y realizar el source local:

- . install/setup.bash

Ahora se puede correr el nodo:

- ros2 run learning_tf2_cpp static_turtle_tf2_broadcaster mystaticturtle 0 0 1 0 0 0

Los valores finales (0 0 1 0 0 0) corresponden a los movimientos en los planos y sus respectivas rotaciones: ‘X’, ‘Y’, ‘Z’, ‘Roll’, Pitch’, ‘Yaw’.

Se puede verificar la transformación estática siendo publicada empleando *echo*:

- ros2 topic echo --qos-reliability reliable --qos-durability transient_local /tf_static

Lo cual debe imprimir en pantalla:

```
transforms:
- header:
  stamp:
    sec: 1622908754
    nanosec: 208515730
  frame_id: world
child_frame_id: mystaticturtle
transform:
  translation:
    x: 0.0
    y: 0.0
    z: 1.0
  rotation:
    x: 0.0
    y: 0.0
    z: 0.0
    w: 1.0
```

The proper way to publish static transforms: Anteriormente se enseñó como se usaba *StaticTransformBroadcaster* para publicar transformaciones estátifcas, pero en un desarrollo real, no se tiene que escribir este código, se puede emplear la herramienta dedicada *tf2_ros* que provee un ejecutable llamado *static_transform_publisher* que puede ser usado como una línea de comando o un nodo que se puede añadir a los *launch files* propios.

Publicación de coordenadas de transformación estáticas usando ‘X’, ‘Y’, ‘Z’, ‘Roll’, Pitch’, ‘Yaw’:

- ros2 run tf2_ros static_transform_publisher x y z yaw pitch roll frame_id child_frame_id

Publicación de coordenadas de transformación estáticas usando ‘X’, ‘Y’, ‘Z’ y un desface.

- ros2 run tf2_ros static_transform_publisher x y z qx qy qz qw frame_id child_frame_id

La otra forma de definir el *static_transform_publisher* es agregándola en el código del nodo o del *launch file*:

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

def generate_launch_description():
    return LaunchDescription([
        Node(
            package='tf2_ros',
            executable='static_transform_publisher',
            arguments = ['0', '0', '1', '0', '0', '0', 'world', 'mystaticturtle']
        ),
    ])
```

Writing a broadcaster: Dentro de la carpeta src del paquete *learninf_tf2_cpp* se debe crear el nodo, para eso se importa desde github:

- wget https://raw.githubusercontent.com/ros/geometry_tutorials/ros2/turtle_tf2_cpp/src/turtle_tf2_broadcaster.cpp

Agregar el siguiente código al CMakeLists.txt:

```
add_executable(turtle_tf2_broadcaster src/turtle_tf2_broadcaster.cpp)
ament_target_dependencies(
  turtle_tf2_broadcaster
  geometry_msgs
  rclcpp
  tf2
  tf2_ros
  turtlesim
)
```

También agregar a la sección de *install* el nodo creado.

```
install(TARGETS
  turtle_tf2_broadcaster
  DESTINATION lib/${PROJECT_NAME})
```

Ahora se debe crear el launch file, para eso crear una carpeta llamada *launch* dentro de la raíz del paquete *src/learning_tf_cpp* y crear un archivo llamado *turtle_tf2_demo.launch.py* con el siguiente código:

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


def generate_launch_description():
  return LaunchDescription([
    Node(
      package='turtlesim',
      executable='turtlesim_node',
      name='sim'
    ),
    Node(
      package='learning_tf2_cpp',
      executable='turtle_tf2_broadcaster',
      name='broadcaster1',
      parameters=[
        {'turtlename': 'turtle1'}
      ]
    ),
```

```
  ])
```

Ahora se agregan las dependencias al archivo *package.xml*:

```
<exec_depend>launch</exec_depend>
<exec_depend>launch_ros</exec_depend>
```

Ahora se nuevamente dentro del *CMakeLists.txt* agregar:

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

Ejecutar el rosdep en la raíz del ws para verificar si faltan dependencias:

- rosdep install -i --from-path src --rosdistro foxy -y

Eecutar el build:

- colcon build --packages-select learning_tf2_cpp

Realizar el source local:

- . install/setup.bash

Ahora se puede ejecutar:

- ros2 launch learning_tf2_cpp turtle_tf2_demo.launch.py

En una nueva terminal ejecutar el teleop:

- ros2 run turtlesim turtle_teleop_keyEn un

En una tercera terminal ejecutar el echo para visualizar los datos de posición que están siendo publicados en el topic:

- ros2 run tf2_ros tf2_echo world turtle1

En el caso del ejemplo en mi computador, yo llamé al `t.header.frame_id = "world";`
es por eso, que en mi caso, el echo se ejecuta:

- ros2 run tf2_ros tf2_echo world turtle1

Writing a listener (C++):

Dentro de la carpeta src del paquete, ejecutar:

- wget https://raw.githubusercontent.com/ros/geometry_tutorials/ros2/turtle_tf2_cpp/src/turtle_tf2_listener.cpp

Esto importará de Git el nodo del listener.

Actualizar el *CMakeLists.txt*:

```
add_executable(turtle_tf2_listener src/turtle_tf2_listener.cpp)
ament_target_dependencies(
  turtle_tf2_listener
  geometry_msgs
  rclcpp
  tf2
  tf2_ros
  turtlesim
)
```

Y en la sección de *install* agregar:

```
install(TARGETS
  turtle_tf2_listener
  DESTINATION lib/${PROJECT_NAME})
```

Se tiene que actualizar el launch file para quedar:

```
from launch import LaunchDescription
from launch.actions import DeclareLaunchArgument
from launch.substitutions import LaunchConfiguration

from launch_ros.actions import Node


def generate_launch_description():
  return LaunchDescription([
    Node(
      package='turtlesim',
      executable='turtlesim_node',
      name='sim'
    ),
    Node(
      package='learning_tf2_cpp',
      executable='turtle_tf2_broadcaster',
      name='broadcaster1',
      parameters=[
        {'turtlename': 'turtle1'}
```

```
            ]
        ),
        DeclareLaunchArgument(
            'target_frame', default_value='turtle1',
            description='Target frame name.'
        ),
        Node(
            package='learning_tf2_cpp',
            executable='turtle_tf2_broadcaster',
            name='broadcaster2',
            parameters=[
                {'turtlename': 'turtle2'}
            ]
        ),
        Node(
            package='learning_tf2_cpp',
            executable='turtle_tf2_listener',
            name='listener',
            parameters=[
                {'target_frame': LaunchConfiguration('target_frame')}
            ]
        ),
    ])
```

Verificar las dependencias:

- rosdep install -i --from-path src --rosdistro foxy -y

Realizar el build:

- colcon build --packages-select learning_tf2_cpp

Realizar el source local:

- . install/setup.bash

Ahora se puede ejecutar. En una terminal:

- ros2 launch learning_tf2_cpp turtle_tf2_demo.launch.py

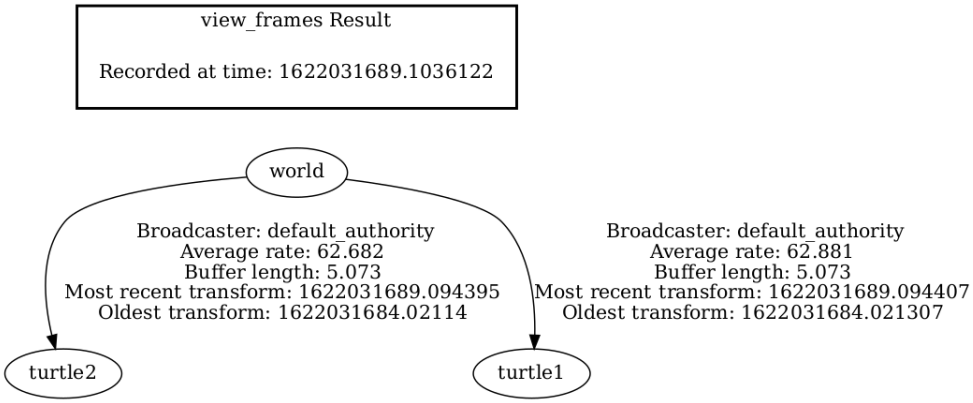
En una segunda terminal:}

- ros2 run turtlesim turtle_teleop_key

Adding a frame (C++): Anteriormente se vio como recrear el *turtle demo* escribiendo un *broadcaster* y un *listener*, en esta ocasión se aprenderá como agregar *frames* extra fijados a los *frames*

dinámicos del árbol de transformación. Añadir un *frame* es muy similiar a la creación del *tf2 broadcaster*, pero en este ejemplo, se mostrarán algunos elementos adicionales.

En el árbol del *tf2*, un *frame* puede tener solo un padre, pero puede tener muchos hijos. Actualmente, nuestro *tf2* contiene tres frames: *world, turtle1, turtle2.* Los dos frames de *turtles* son los hijos del *frame* padre *world*.



Write the fixed broadcaster: En el ejempo de la tortuga se añadirá un frame llamado *carrot1*, que será un hijo de *turtle1*, este frame será el objetivo de la segunda tortuga.
Dentro del paquete creado ejecutar:

- wget https://raw.githubusercontent.com/ros/geometry_tutorials/ros2/turtle_tf2_cpp/src/fixed_frame_tf2_broadcaster.cpp

Esto creará un archivo llamado *fixed_frame_tf2_broadcaster.cpp*.

```
geometry_msgs::msg::TransformStamped t;

t.header.stamp = this->get_clock()->now();
t.header.frame_id = "turtle1";
t.child_frame_id = "carrot1";
t.transform.translation.x = 0.0;
t.transform.translation.y = 2.0;
t.transform.translation.z = 0.0;
```

La mayor diferencia entre el *broadcaster* de la tortuga 1, es que las transformaciones de posición no cambian en el tiempo (vistas desde el frame padre), en este caso, *carrot1* se ubica a dos metros en el eje ‘y’ desde la posición del padre (*turtle1*).

Se agrega el ejecutable en el *CMakeLists.txt*:

```
add_executable(fixed_frame_tf2_broadcaster src/fixed_frame_tf2_broadcaster.cpp)
ament_target_dependencies(
  fixed_frame_tf2_broadcaster
  geometry_msgs
  rclcpp
  tf2_ros
)
```

Y se agrega el *fixed_frame_tf2_broadcaster* a la parte de install dentro del *CMakeLists.txt*:

```
install(TARGETS
  fixed_frame_tf2_broadcaster
  DESTINATION lib/${PROJECT_NAME})
```

Ahora se creará un *launch file* para este nuevo *frame*, el nuevo archivo será llamado *turtle_tf2_fixed_frame_demo.launch.py* y tendrá el siguienre código:

```
import os

from ament_index_python.packages import get_package_share_directory

from launch import LaunchDescription
from launch.actions import IncludeLaunchDescription
from launch.launch_description_sources import PythonLaunchDescriptionSource

from launch_ros.actions import Node


def generate_launch_description():
  demo_nodes = IncludeLaunchDescription(
    PythonLaunchDescriptionSource([os.path.join(
      get_package_share_directory('learning_tf2_cpp'), 'launch'),
      '/turtle_tf2_demo.launch.py']),
    )

  return LaunchDescription([
    demo_nodes,
    Node(
      package='learning_tf2_cpp',
      executable='fixed_frame_tf2_broadcaster',
      name='fixed_broadcaster',
    ),
  ])
```

Se puede notar que este *launch file* llama al *launch file* creado anteriormente (que crea un mundo con dos tortugas en donde *turtle2* sigue la posición de *turtle1*).

Se verifican las rosdep:

- rosdep install -i --from-path src --rosdistro foxy -y

Se realiza el build:

- colcon build --packages-select learning_tf2_cpp

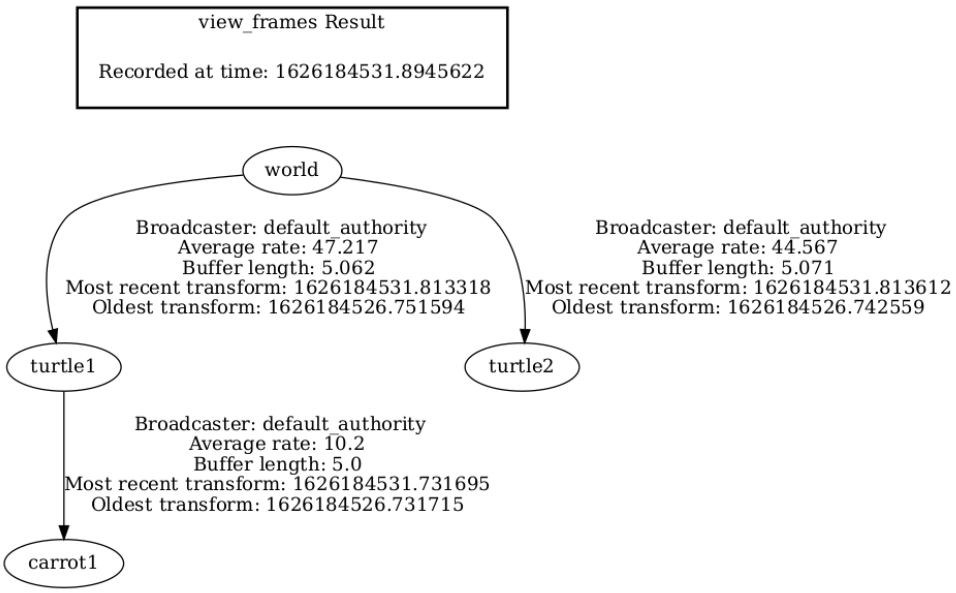
Luego, se realiza el source local:

- . install/setup.bash

Ahora se puede correr la demo:

- ros2 launch learning_tf2_cpp turtle_tf2_fixed_frame_demo.launch.py

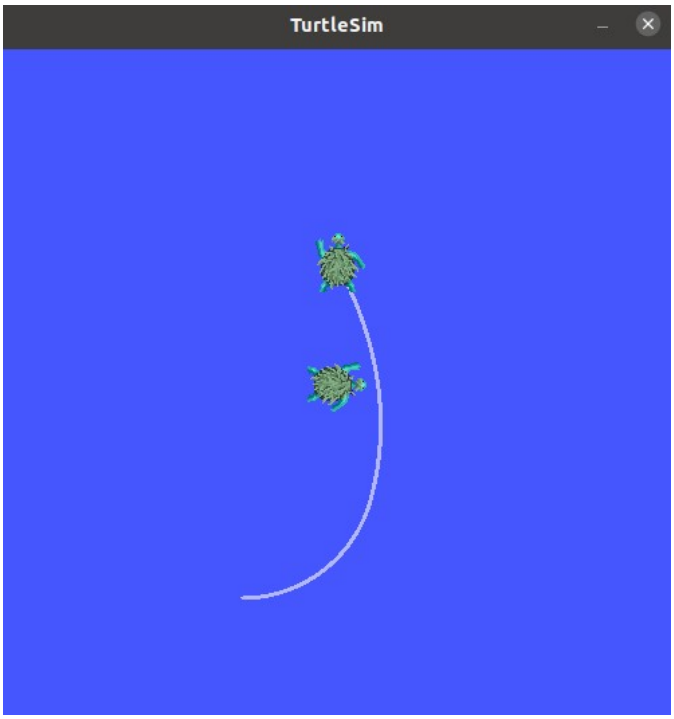
El árbol se ve ahora así:



Se puede notar que el comportamiento de las tirtugas no cambia añadiendo un nuevo frame, no obstante, si se quiere cambiar el objetivo de la tortuga2 para que siga a *carrot1* en vez de a *turtle1*, se debe ejecutar:

- ros2 launch learning_tf2_cpp turtle_tf2_fixed_frame_demo.launch.py target_frame:=carrot1

Realizando este cambio, se puede notar como la tortuga2 sigue el *frame* de *carrot1* que se encuentra en una ubicación de +2 metros en el eje ‘y’.



Una segunda forma de cambiar el objetivo de la tortuga2 es desde el archivo *turtle_tf2_fixed_frame_demo.launch.py*, cambiando el parámetro de *'target_frame'* tal que:

- 'target_frame': 'carrot1'

Quedaría así:

```
def generate_launch_description():
    demo_nodes = IncludeLaunchDescription(
        ...,
        launch_arguments={'target_frame': 'carrot1'}.items(),
        )
```

Si se hace esto, se tiene que volver a hacer el procedimiento del rosdep, build y source local.

Write the dynamic frame broadcaster: El *frame* extra que se agregó anteriormente, estaba fijo respecto a su *frame* padre, si se quiere crear un nuevo *frame* que cambie a lo largo del tiempo respecto a su padre, dentro de la carpeta src del paquete creado ejecutar:

- wget https://raw.githubusercontent.com/ros/geometry_tutorials/ros2/turtle_tf2_cpp/src/dynamic_frame_tf2_broadcaster.cpp

Esto dejará un archivo llamado *dynamic_frame_tf2_broadcaster.cpp* en el cual la transformación de la posición es una función:

```
double x = now.seconds() * PI;
...
t.transform.translation.x = 10 * sin(x);
t.transform.translation.y = 10 * cos(x);
```

Se debe agregar el ejecutable al archivo *CMakeLists.txt*:

```
add_executable(dynamic_frame_tf2_broadcaster src/dynamic_frame_tf2_broadcaster.cpp)
ament_target_dependencies(
  dynamic_frame_tf2_broadcaster
  geometry_msgs
  rclcpp
  tf2_ros
)
```

Ahora se debe agregar en el apartado de *install*:

```
install(TARGETS
  dynamic_frame_tf2_broadcaster
  DESTINATION lib/${PROJECT_NAME})
```

Para probar este nuevo código, se crea un archivo llamado *turtle_tf2_dynamic_frame_demo.launch.py* el cual contiene el siguiente código:

```
import os

from ament_index_python.packages import get_package_share_directory

from launch import LaunchDescription
from launch.actions import IncludeLaunchDescription
from launch.launch_description_sources import PythonLaunchDescriptionSource

from launch_ros.actions import Node


def generate_launch_description():
  demo_nodes = IncludeLaunchDescription(
    PythonLaunchDescriptionSource([os.path.join(
      get_package_share_directory('learning_tf2_cpp'), 'launch'),
      '/turtle_tf2_demo.launch.py']),
    launch_arguments={'target_frame': 'carrot1'}.items(),
```

```
    )

  return LaunchDescription([
    demo_nodes,
    Node(
      package='learning_tf2_cpp',
      executable='dynamic_frame_tf2_broadcaster',
      name='dynamic_broadcaster',
    ),
  ])
```

Se verifican las rosdep:

- rosdep install -i --from-path src --rosdistro foxy -y

Se realiza el build:

- colcon build --packages-select learning_tf2_cpp

Luego, se realiza el source local:

- . install/setup.bash

Ahora se puede correr la demo:

- ros2 launch learning_tf2_cpp turtle_tf2_dynamic_frame_demo.launch.py

Using time (C++): Anteriormente, se recreó la *demo* de la tortuga escribiendo un nodo *tf2 broadcaster* y uno *tf2 listener* y se aprendió como *tf2* hacía un seguimiento del árbol jerárquico de los *frames* de coordenadas. Este árbol cambia a través del tiempo y guarda información de cada transformación hasta por 10 segundos (por defecto); hasta ahora se ha empleado la función *lookupTransform* (verificar el código de los nodos) para tener acceso al la última transformación disponible en el árbol de *tf2*, pero sin conoves a qué tiempo era grabada. Para obtener una transformación en un tiempo específico. Dirigirse al archivo *turtle_tf2_listener.cpp* que se encuentra dentro del paquete *learning_tf2cpp*, allí se podrá ver que en la función *lookupTransform()* se tiene *tf2:TimePonitZero*, esto significa que se está especificando que el tiempo es igual a cero, paraw*tf2* un tiempo cero significa “la última disponible”. En *tf2* es posible viajar en el tiempo, es decir, una característica de *tf2* es que es una librería capaz de transformar datos tanto en espacio como en tiempo.

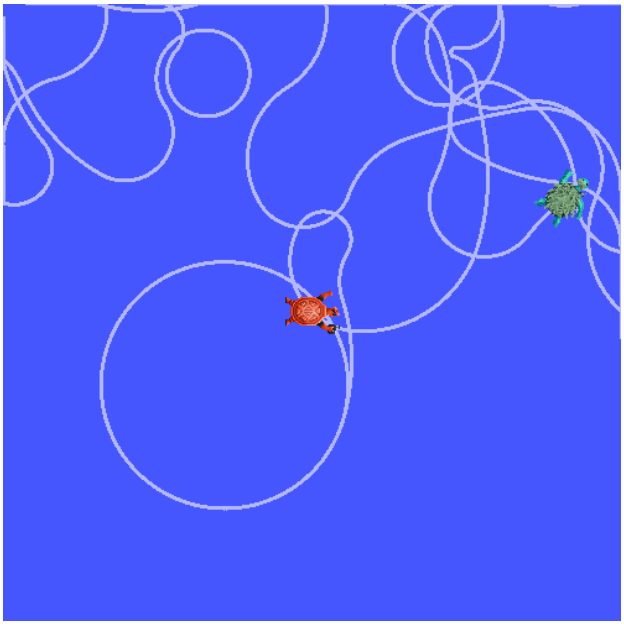
Esta característica de viajar en el tiempo es bastante útil si se quiere monitorear la posición de un robot a lo largo del tiempo, o construir un robot segidor que “siga” los pasos de su lider. Para tener un ejemplo, volver a abrir el archivo mencionado en el párrafo de arriba, y editar la función *lookupTransform()*:

```
rclcpp::Time when = this->get_clock()->now() - rclcpp::Duration(5, 0);
t = tf_buffer_->lookupTransform(
  toFrameRel,
  fromFrameRel,
  when,
```

```
    50ms);
```

Se puede notar que también se agregó una línea de código antes de la declaración de la función *lookupTrnansfor()*, esta línea adicional se emplea para obtener datos en “tiempo real”, sin embargo, no permitirá que se publique información sin tener 5 segundos de datos históricos de la posición de la zanahoria (*carrot1*) que se declaró en páginas anteriores. Pero, ¿qué pasa después?
Ejecutar en una terminal:

- ros2 launch learning_tf2_cpp turtle_tf2_fixed_frame_demo.launch.py



Se podrá notar que la tortuga estará realizando movimientos descontrolados sin saber a donde dirigirse, pero vamos a entender las razones de este comportamiento:

En el código, se hicieron las siguientes preguntas: ¿Cuál fue posición de *carrot1* hace 5 segundos, y ¿Cuál fue su posición relativa con *turtle2* hace 5 segundos?. Esto significa, que se está controlando la segunda tortuga respecto a su posición de 5 segundos atrás y la posición relativa de la zanahoria 5 segundos en el pasado. Aunque suena loca la idea, en todo momento se le está enviando una información de la posición de la zanahoria cinco segundos atrás, en relación con su posición 5 segundos atrás, es decir, que se calcula su posición actual con datos que ya no “tienen sentido físico” pues la tortuga nunca será capaz de conocer su posición ya que está en una simulación del presente, pero con información de un mundo pasado.

Advanced API for lookupTransform(): Existe una API que otorga el poder de decir explícitamente cuando obtener las transformaciones especificadas, esto se hace llamando al método *lookupTransform()* con parámetros adicionales:

```
rclcpp::Time now = this->get_clock()->now();
rclcpp::Time when = now - rclcpp::Duration(5, 0);
t = tf_buffer_->lookupTransform(
    toFrameRel,
```

```
now,
fromFrameRel,
when,
"world",
50ms);
```

Las API avanzadas para *lookupTransform()* tiene seis argumentos:

1. *Frame* objetivo.
2. El tiempo de la transformación.
3. EL *frame* seguidor.
4. El tiempo en el cual el *frame* seguidor será evaluado.
5. Un frame que no cambia a lo largo del tiempo

Si se ejecuta la simulación, se verá que la tortuga se dirige a la posición que tenía la zanahoria 5 segundos atrás. Esto pasa, porque definí un tiempo pasado y el tiempo actual, por eso es posible realizar las transformaciones.

Quartenions: ROSusa los cuartenios *quartenions* para seguir y aplicar rotaciones. Un cuarternio tiene 4 componentes (x, y, z, w). En ROS 2 'w' está de último, pero en algunas librerías como *Eigen*, 'w' se posiciona de primero. Un cuartenio que no contiene ninguna rotación, se escribe (0, 0, 0, 1), y puede ser creado de la siguiente forma:

```
#include <tf2/LinearMath/Quaternion.h>
...

tf2::Quaternion q;
// Create a quaternion from roll/pitch/yaw in radians (0, 0, 0)
q.setRPY(0, 0, 0);
// Print the quaternion components (0, 0, 0, 1)
RCLCPP_INFO(this->get_logger(), "%f %f %f %f",
        q.x(), q.y(), q.z(), q.w());
```

La magnitud de un cuartenio debería ser siempre 1, en caso de que no sea así, se imprimirá una advertencia, para evitar esto, se debe normalizar el cuartenio:

```
q.normalize();
```

ROS 2 usa dos tipos de datos para cuartenios:

1. tf2::Quartenion
2. geometry_msgs::msg::Quartenion

Para hacer conversiones entre ellos empleando C++, se escribe en el código:

```
#include <tf2_geometry_msgs/tf2_geometry_msgs.hpp>
#include <tf2/LinearMath/Quaternion.h>

...
```

```
tf2::Quaternion tf2_quat, tf2_quat_from_msg;
tf2_quat.setRPY(roll, pitch, yaw);
// Convert tf2::Quaternion to geometry_msgs::msg::Quaternion
geometry_msgs::msg::Quaternion msg_quat = tf2::toMsg(tf2_quat);

// Convert geometry_msgs::msg::Quaternion to tf2::Quaternion
tf2::convert(msg_quat, tf2_quat_from_msg);
// or
tf2::fromMsg(msg_quat, tf2_quat_from_msg);
```

Para aplicar la rotación de un cuartenio a una posición, simplemente multiplica el cuartenio anterior de la posición por el cuartenio que representa la rotación final. EL ORDEN DE ESAS MULTIPLICACIONES IMPORTA.

```
#include <tf2_geometry_msgs/tf2_geometry_msgs.hpp>
#include <tf2/LinearMath/Quaternion.h>
...

tf2::Quaternion q_orig, q_rot, q_new;

q_orig.setRPY(0.0, 0.0, 0.0);
// Rotate the previous pose by 180* about X
q_rot.setRPY(3.14159, 0.0, 0.0);
q_new = q_rot * q_orig;
q_new.normalize();
```

Para invertir un cuartenio, basta con agregar un menos (-):

- q[3] = -q[3]

Rotaciones relativas: Supongamos que se tienen dos cuarteniones del mismo *frame*, *q_1* y *q_2*, si se quiere obtener la rotación relatica *q_r* que convierte *q_1* a *q_2*:

- $q\_2 = q\_r * q_1$

Aquí se tiene un ejemplo en Python para obtener rotaciones relativas:

```
def quaternion_multiply(q0, q1):
    """
    Multiplies two quaternions.

    Input
    :param q0: A 4 element array containing the first quaternion (q01, q11, q21, q31)
    :param q1: A 4 element array containing the second quaternion (q02, q12, q22, q32)

    Output
    :return: A 4 element array containing the final quaternion (q03,q13,q23,q33)
```

```
    """
    # Extract the values from q0
    w0 = q0[0]
    x0 = q0[1]
    y0 = q0[2]
    z0 = q0[3]

    # Extract the values from q1
    w1 = q1[0]
    x1 = q1[1]
    y1 = q1[2]
    z1 = q1[3]

    # Computer the product of the two quaternions, term by term
    q0q1_w = w0 * w1 - x0 * x1 - y0 * y1 - z0 * z1
    q0q1_x = w0 * x1 + x0 * w1 + y0 * z1 - z0 * y1
    q0q1_y = w0 * y1 - x0 * z1 + y0 * w1 + z0 * x1
    q0q1_z = w0 * z1 + x0 * y1 - y0 * x1 + z0 * w1

    # Create a 4 element array containing the final quaternion
    final_quaternion = np.array([q0q1_w, q0q1_x, q0q1_y, q0q1_z])

    # Return a 4 element array containing the final quaternion (q02,q12,q22,q32)
    return final_quaternion

q1_inv[0] = prev_pose.pose.orientation.x
q1_inv[1] = prev_pose.pose.orientation.y
q1_inv[2] = prev_pose.pose.orientation.z
q1_inv[3] = -prev_pose.pose.orientation.w # Negate for inverse

q2[0] = current_pose.pose.orientation.x
q2[1] = current_pose.pose.orientation.y
q2[2] = current_pose.pose.orientation.z
q2[3] = current_pose.pose.orientation.w

qr = quaternion_multiply(q2, q1_inv)
```

Using stamped datatypes: Para aprender a usar los datos de un sensor con tf2, se supondrá que se tiene una tercera tortuga *turtle3* que no tiene una buena odometría, pero hay una cámara externa que está siguiendo su posición y publicándola como un mensaje *PointStamped* en relación con el *frame* del mundo (*world*). La tortuga uno desea saber donde está la tortuga 3 comparada con su posición, para esto, *turtle1* debe "escuchar" un *topic* donde la posición de *turtle3* está siendo publicada, esperar hasta que las transformaciones estén listas y luego realizar sus operaciones. Para que sea más sencillo, la herramienta *tf2_ros::MessageFilter* es muy útil, pues toma una suscripción de cualquier mensaje y lo almacena hasta que sea posible transformarlo en el *frame* objetivo. El desarrollo se hará en nodos escritos en Python, se debe crear un paquete llamado *learning_tf2_py*:

- ros2 pkg create --build-type ament_python learning_tf2_py --dependencies geometry_msgs rclcpp tf2 tf2_ros turtlesim

En el directorio *src/learning_tf2_py/learning_tf2_py* importar de GitHub:

- wget https://raw.githubusercontent.com/ros/geometry_tutorials/ros2/turtle_tf2_py/turtle_tf2_py/turtle_tf2_message_broadcaster.py

Ahora se debe escribir el *launch file*, crear una carpeta *launch* y dentro de ella un archivo llamado *turtle_tf2_sensormessage.launch.py* y pegar el siguiente código:

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


def generate_launch_description():
    return LaunchDescription([
        DeclareLaunchArgument(
            'target_frame', default_value='turtle1',
            description='Target frame name.'
        ),
        Node(
            package='turtlesim',
            executable='turtlesim_node',
            name='sim',
            output='screen'
        ),
        Node(
            package='turtle_tf2_py',
            executable='turtle_tf2_broadcaster',
            name='broadcaster1',
            parameters=[
                {'turtlename': 'turtle1'}
            ]
        ),
        Node(
            package='turtle_tf2_py',
            executable='turtle_tf2_broadcaster',
            name='broadcaster2',
            parameters=[
                {'turtlename': 'turtle3'}
            ]
        ),
        Node(
            package='turtle_tf2_py',
            executable='turtle_tf2_message_broadcaster',
            name='message_broadcaster',
        ),
```

```
    ])
```

Ahora en el archivo *setup.py* agregar la siguiente línea dentro de las llaves de *'console_scripts'*:

- 'turtle_tf2_message_broadcaster = learning_tf2_py.turtle_tf2_message_broadcaster:main',

Verificar las rosdep:

- rosdep install -i --from-path src --rosdistro foxy -y

Realizar el build:

- colcon build --packages-select learning_tf2_py

Writing the message filter/listener node: Ahora para obtener los datos de *PointStamped* de *turtle3* en el *frame* de *turtle1*, se debe crear un *source file* del mensaje *filter/listener node*. Ahora hay que dirigirse al paquete *learning_tf2_cpp* y dentro de la carpeta *src* importar de GitHub:

- wget https://raw.githubusercontent.com/ros/geometry_tutorials/ros2/turtle_tf2_cpp/src/turtle_tf2_message_filter.cpp

Se agregan las dependencias en el archivo *package.xml* dentro del paquete actual:

- <depend>message_filters</depend>
- <depend>tf2_geometry_msgs</depend>

Agregar al archivo *CMakeLists.txt*:

- find_package(message_filters REQUIRED)
- find_package(tf2_geometry_msgs REQUIRED)

Más abajo agregar:

```
if(TARGET tf2_geometry_msgs::tf2_geometry_msgs)
  get_target_property(_include_dirs tf2_geometry_msgs::tf2_geometry_msgs
INTERFACE_INCLUDE_DIRECTORIES)
else()
  set(_include_dirs ${tf2_geometry_msgs_INCLUDE_DIRS})
endif()

find_file(TF2_CPP_HEADERS
  NAMES tf2_geometry_msgs.hpp
  PATHS ${_include_dirs}
  NO_CACHE
  PATH_SUFFIXES tf2_geometry_msgs
)
```

Luego, se agrega el ejecutable llamado *turtle_tf2_message_filter* que se usará más adelante:

```
add_executable(turtle_tf2_message_filter src/turtle_tf2_message_filter.cpp)
```

```
ament_target_dependencies(
  turtle_tf2_message_filter
  geometry_msgs
  message_filters
  rclcpp
  tf2
  tf2_geometry_msgs
  tf2_ros
)

if(EXISTS ${TF2_CPP_HEADERS})
  target_compile_definitions(turtle_tf2_message_filter PUBLIC -DTF2_CPP_HEADERS)
endif()
```

Por último, agregar el nodo al apartado de *install (TARGETS…*

```
install(TARGETS
  turtle_tf2_message_filter
  DESTINATION lib/${PROJECT_NAME})
```

Verificar las rosdep:

- rosdep install -i --from-path src --rosdistro foxy -y

Realizar el build:

- colcon build --packages-select learning_tf2_py

En una nueva terminal realizar un source local y ejecutar:

- ros2 launch learning_tf2_py turtle_tf2_sensor_message.launch.py

En una nueva terminal, realizar un source y ejecutar el nodo de teleop:

- ros2 run turtlesim turtle_teleop_key

URDF

Para poder usar urdf, primero se deben tener instaladas algunas dependencias, en una terminal realizar un source general y ejecutar:

- sudo apt install ros-foxy-joint-state-publisher-gui
- sudo apt install ros-foxy-xacro

Dentro de un ws ya creado, se debe agregar un paquete que contendrá todo lo relacionado al urdf del proyecto:

- ros2 pkg create --build-type ament_cmake car_description

En este caso, el nombre del paquete será *car_desciption*, sin embargo, el nombre puede elegirse a conveniencia. Dentro del *src* del paquete, crear una carpeta *description* y un archivo dentro de ese directorio llamado *car.urdf* que inicialmente contenga lo siguiente:

```
<?xml version="1.0"?>
	<robot name="car" xmlns:xacro="http://ros.org/wiki/xacro">



</robot>
```

Dentro de los “tags” de <robot> se van a agregar algunas constantes empleando *xacro:property*:

```
<?xml version="1.0"?>

<robot name="car" xmlns:xacro="http://ros.org/wiki/xacro">

	<!-- Define robot constants -->
	<xacro:property name="base_width" value="0.120"/>
	<xacro:property name="base_length" value="0.200"/>
	<xacro:property name="base_height" value="0.055"/>

	<xacro:property name="wheel_radius" value="0.0375"/>
	<xacro:property name="wheel_width" value="0.030"/>
	<xacro:property name="wheel_ygap" value="0.005"/>
	<xacro:property name="wheel_xoff" value="0.1125"/>
	<xacro:property name="wheel_zoff" value="0.0275"/>



</robot>
```

Las propiedades de *base_* hacer referencia a las dimensiones del chasis del robot, *wheel_radius* & *wheel_width* definen la forma de las ruedas, *wheel_ygap* define un *offset* en el eje y de las ruedas respecto al chasis. Ahora se definirá el *base_link*, este será una caja alargada que funcionará como el chasis. En URDF un *link* representa un componente rígido del robot, el *state publisher* utiliza esta definición para determinar los *frames* de cada *link* y publicar la transformación entre ellos.

```
<!-- Robot Base -->
<link name="base_link">
	<visual>
		<geometry>
			<box size="${base_length} ${base_width} ${base_height}"/>
		</geometry>
```

```
            <material name="Cyan">
                <color rgba="0 1.0 1.0 1.0"/>
            </material>
        </visual>
</link>
```

Ahora se definirá un *base_footprint* que es un link no físico que no tienen propiedades de colisión ni dimensiones, pero es necesario para poder habilitar varios paquetes, esto determina el centro de la sombra del robot proyectada en el suelo. Por ejemplo, *Navigation2* emplea este link para poder calcular la evasión de obstáculos.
En URDF, un *joint* describe propiedades cinemáticas y dinámicas entre *frames*. En este caso, el *joint* será de tipo *fixed* (fijo) con el respectivo offset para que cumpla las condiciones de estar proyectado en el suelo.

```
<!-- Robot Footprint -->
<link name="base_footprint"/>

<joint name="base_joint" type="fixed">
        <parent link="base_link"/>
        <child link="base_footprint"/>
        <origin xyz="0.0 0.0 ${-(wheel_radius+wheel_zoff)}" rpy="0 0 0"/>
</joint>
```

Se puede notar, que el desface en el eje ‘z’ corresponde al radio de las ruedas sumado al desaface de éstas en el eje ‘z’, no obstante, en el robot que se está describiendo, no existe tal desfase, por lo tanto, debe eliminar *wheel_zoff* del código.

Para agregar las llantas y que el código no quede tedioso y extenso, se emplean macros. Se define la macro la cuál tiene como parámetros un prejifo (*prefix*), un reflejo en x y otro reflejo en y; los reflejos sirven para poder determinar la ubicación de las ruedas solamente multiplicando por ±1 las constantes definidas anteriormente. El *joint* de tipo *continuos* permite generar un movimiento rotacional continuo (como una rueda). La ubicación de las ruedas se define de la siguiente forma:

1. En el eje ‘x’ es el producto entre *x_reflect* y *wheel_xoff*, esto es porque el desfase *wheel_xoff* corresponde a ½ de la distancia entre ejes, y al multiplicarlo por el factor de reflejo, este valor será en la dirección ±x, pudiendo así ubicar las ruedas delanteras y traseras.
2. El eje ‘y’

```
<!-- Wheels -->
<xacro:macro name="wheel" params="prefix x_reflect y_reflect">
<link name="${prefix}_link">
        <visual>
            <origin xyz="0 0 0" rpy="${pi/2} 0 0"/>
            <geometry>
                <cylinder radius="${wheel_radius}" length="${wheel_width}"/>
            </geometry>
            <material name="Gray">
                <color rgba="0.5 0.5 0.5 1.0"/>
```

```
                </material>
        </visual>
</link>

<joint name="${prefix}_joint" type="continuous">
        <parent link="base_link"/>
        <child link="${prefix}_link"/>
        <origin xyz="${x_reflect*wheel_xoff} ${y_reflect*(base_width/2+wheel_ygap)} ${-
        wheel_zoff}" rpy="0 0 0"/>
        <axis xyz="0 1 0"/>
</joint>
</xacro:macro>

<xacro:wheel prefix="front_wheelR" x_reflect="0.8" y_reflect="-1" />
<xacro:wheel prefix="front_wheelL" x_reflect="0.8" y_reflect="1" />
<xacro:wheel prefix="back_wheelR" x_reflect="-1" y_reflect="-1" />
<xacro:wheel prefix="back_wheelL" x_reflect="-1" y_reflect="1" />
```

Se puede notar que las ruedas delanteras *front_wheel* tienen en *x_reflect* un valor de 0.8, esto es porque en la construcción del vehículo, las ruedas traseras están un poco más alejadas del centro del chasis comparadas con las ruedas delanteras, la relación es de aproximadamente 0.8.

Ahora que ya está finalizada la descripción del robot, se deben agregar las dependencias en el archivo *package.xml*

```
<exec_depend>joint_state_publisher</exec_depend>
<exec_depend>joint_state_publisher_gui</exec_depend>
<exec_depend>robot_state_publisher</exec_depend>
<exec_depend>rviz</exec_depend>
<exec_depend>xacro</exec_depend>
```

Se procede a crear el *launch file* para poder ejecutar y visualizar la descripción del robot. Se debe crear una carpeta llamada *launch* y dentro de ella un archivo que lleva el nombre de *display.launch.py*

```
import launch
from launch.substitutions import Command, LaunchConfiguration
import launch_ros
import os

def generate_launch_description():
pkg_share =
launch_ros.substitutions.FindPackageShare(package='car_description').find('car_description')
default_model_path = os.path.join(pkg_share, 'src/description/car.urdf')
default_rviz_config_path = os.path.join(pkg_share, 'rviz/urdf_config.rviz')

robot_state_publisher_node = launch_ros.actions.Node(
package='robot_state_publisher',
executable='robot_state_publisher',
```

```
parameters=[{'robot_description': Command(['xacro ', LaunchConfiguration('model')])}]
)
joint_state_publisher_node = launch_ros.actions.Node(
package='joint_state_publisher',
executable='joint_state_publisher',
name='joint_state_publisher',
condition=launch.conditions.UnlessCondition(LaunchConfiguration('gui'))
)
joint_state_publisher_gui_node = launch_ros.actions.Node(
package='joint_state_publisher_gui',
executable='joint_state_publisher_gui',
name='joint_state_publisher_gui',
condition=launch.conditions.IfCondition(LaunchConfiguration('gui'))
)
rviz_node = launch_ros.actions.Node(
package='rviz2',
executable='rviz2',
name='rviz2',
output='screen',
arguments=['-d', LaunchConfiguration('rvizconfig')],
)

return launch.LaunchDescription([
launch.actions.DeclareLaunchArgument(name='gui', default_value='True',
description='Flag to enable joint_state_publisher_gui'),
launch.actions.DeclareLaunchArgument(name='model', default_value=default_model_path,
description='Absolute path to robot urdf file'),
launch.actions.DeclareLaunchArgument(name='rvizconfig',
default_value=default_rviz_config_path,
description='Absolute path to rviz config file'),
joint_state_publisher_node,
joint_state_publisher_gui_node,
robot_state_publisher_node,
rviz_node
])
```

Para facilitar la visualización, se creará un archivo que contenga las configuraciones iniciales del Rviz para una correcta ejecución. Dentro del paquete, se crea una carpeta llamada *rviz* que contiene un archivo con el nombre de *urdf_config.rviz* y contiene:

```
Panels:
  - Class: rviz_common/Displays
    Help Height: 78
    Name: Displays
    Property Tree Widget:
      Expanded:
        - /Global Options1
        - /Status1
```

```
      - /RobotModel1/Links1
      - /TF1
    Splitter Ratio: 0.5
  Tree Height: 557
Visualization Manager:
  Class: ""
  Displays:
    - Alpha: 0.5
      Cell Size: 1
      Class: rviz_default_plugins/Grid
      Color: 160; 160; 164
      Enabled: true
      Name: Grid
    - Alpha: 0.6
      Class: rviz_default_plugins/RobotModel
      Description Topic:
        Depth: 5
        Durability Policy: Volatile
        History Policy: Keep Last
        Reliability Policy: Reliable
        Value: /robot_description
      Enabled: true
      Name: RobotModel
      Visual Enabled: true
    - Class: rviz_default_plugins/TF
      Enabled: true
      Name: TF
      Marker Scale: 0.3
      Show Arrows: true
      Show Axes: true
      Show Names: true
  Enabled: true
  Global Options:
    Background Color: 48; 48; 48
    Fixed Frame: base_link
    Frame Rate: 30
  Name: root
  Tools:
    - Class: rviz_default_plugins/Interact
      Hide Inactive Objects: true
    - Class: rviz_default_plugins/MoveCamera
    - Class: rviz_default_plugins/Select
    - Class: rviz_default_plugins/FocusCamera
    - Class: rviz_default_plugins/Measure
      Line color: 128; 128; 0
  Transformation:
    Current:
      Class: rviz_default_plugins/TF
  Value: true
```

```
  Views:
    Current:
      Class: rviz_default_plugins/Orbit
      Name: Current View
      Target Frame: <Fixed Frame>
      Value: Orbit (rviz)
    Saved: ~
```

Por último, se agregará dentro de *CMakeLists.txt*:

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

Verificar las rosdep:

- rosdep install -i --from-path src --rosdistro foxy -y

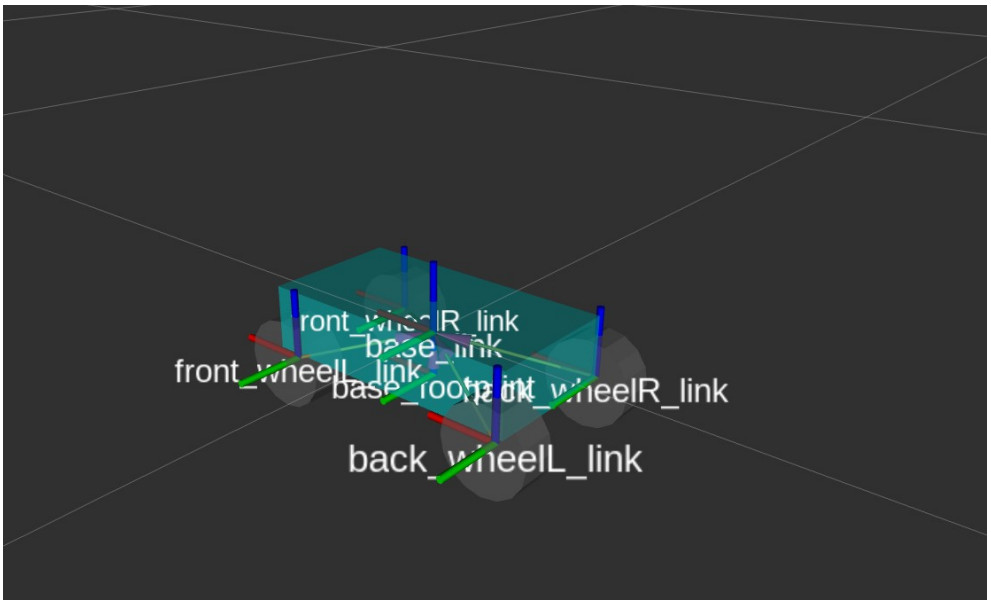
Realizar el build:

- colcon build

En una nueva terminal realizar un source local y ejecutar:

- ros2 launch car_description display.launch.py

El robot quedaría:



Ahora se agregarán las propiedades físicas al robot, para que pueda interactuar con diferentes objetos dentro del entorno de simulación. Se agrega al *car.urdf*

```
<!-- Define intertial property macros -->
<xacro:macro name="box_inertia" params="m w h d">
<inertial>
<origin xyz="0 0 0" rpy="${pi/2} 0 ${pi/2}"/>
<mass value="${m}"/>
<inertia ixx="${(m/12) * (h*h + d*d)}" ixy="0.0" ixz="0.0" iyy="${(m/12) * (w*w + d*d)}"
iyz="0.0" izz="${(m/12) * (w*w + h*h)}"/>
</inertial>
</xacro:macro>

<xacro:macro name="cylinder_inertia" params="m r h">
<inertial>
<origin xyz="0 0 0" rpy="${pi/2} 0 0" />
<mass value="${m}"/>
<inertia ixx="${(m/12) * (3*r*r + h*h)}" ixy = "0" ixz = "0" iyy="${(m/12) * (3*r*r + h*h)}"
iyz = "0" izz="${(m/2) * (r*r)}"/>
</inertial>
</xacro:macro>
```

Las propiedades definidas, corresponden con los momentos de inercia de sobre cada eje de los elementos geométricos que componen el robot descrito.

Ahora se deben agregar los tagas de <*collision*> junto con las propiedades de las macros creadas.

```
<collision>
        <geometry>
                <box size="${base_length} ${base_width} ${base_height}"/>
        </geometry>
</collision>

<xacro:box_inertia m="15" w="${base_width}" d="${base_length}" h="${base_height}"/>
```

```
<collision>
        <origin xyz="0 0 0" rpy="${pi/2} 0 0"/>
        <geometry>
                <cylinder radius="${wheel_radius}" length="${wheel_width}"/>
        </geometry>
</collision>
```

- rosdep install -i --from-path src --rosdistro foxy -y

Realizar el build:

- colcon build

En una nueva terminal realizar un source local y ejecutar:

- ros2 launch car_description display.launch.py